Universiteit Leiden
Opleiding Informatica

Sparse Voxel Mip-Maps:

Versatile Lossy Compression of Volumetric Data

Name:	Micky E. Faas
Date:	29/08/2016
Supervisor:	Dr. Kristian Rietveld
Supervisor:	Dr. Ir. Fons J. Verbeek

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Sparse Voxel Mip-Maps: Versatile Lossy Compression of Volumetric Data

Micky E. Faas, Dr. Kristian Rietveld, Dr. Ir. Fons J. Verbeek

Abstract

Volumetric data is widely used in scientific and medical applications and now also gains popularity in the games and CGI industry, owing to the many advantages it holds over a polygonal representation. Several highly performant discrete raycast algorithms are now available, not in the least thanks to the benefits of GPGPU computing. The large demands on storage and bandwidth, however, may pose a significant limitation on the exchange or (networked) rendering of larger datasets. Despite the fact that there many efficient compression algorithms (mostly based on either Sparse Voxel Octrees (SVO) or Run-Length Encoding (RLE)), almost all of these algorithms are designed to be loss-less, limiting their maximum potential compression ratio. This paper proposes an all-round lossy compression algorithm based on MIP[1] maps, SVOs and block compression. Our algorithm can be used for a variety of applications and voxel formats, both on disk and in memory. It shows a promising compression ratio while maintaining a balanced image quality and traversal speed. Futhermore, an open-source proof-of-concept library is provided as well as a traversal implementation in CUDA.

[1] MIP, in this context, stands for *multum in parvo* meaning 'much in little'

Contents

Chapter 1

Introduction and motivation

Voxels are the future. Volume rendering - the process of visualising voxelized data - is a big topic and has increasingly become so over the recent years. Besides the recent rebirth of volume-based rendering in games, voxelized graphics are now ubiquitous in the special effects industry as well. The idea, however, is as old as computer graphics itself and has been pivotal in the visualisation of captured data for decades. In this first chapter we will start to outline some of the major benefits and obstacles in the world of *direct volume rendering*[1], as well as introduce the scope, scale and goals of this particular research.

1.1 Motivation

The compression and rendering algorithm that is presented in this thesis, is part of a larger project that is being developped at the Leiden Institute for Advanced Computer Science (LIACS). The aim is to provide a complete solution for preprocessing, storing, rendering and interacting with volumetric data sets. The main focus lies on imaging sets obtained from various microscopy scanning techniques and its potential user base would be, for example, the department of Imaging and Bioinformatics. The challenge that sets this project apart from other visualisation projects, is the fact that it is targeted at a very high-resolution display array, comparable to the work by Schwarz et al. [SL10].

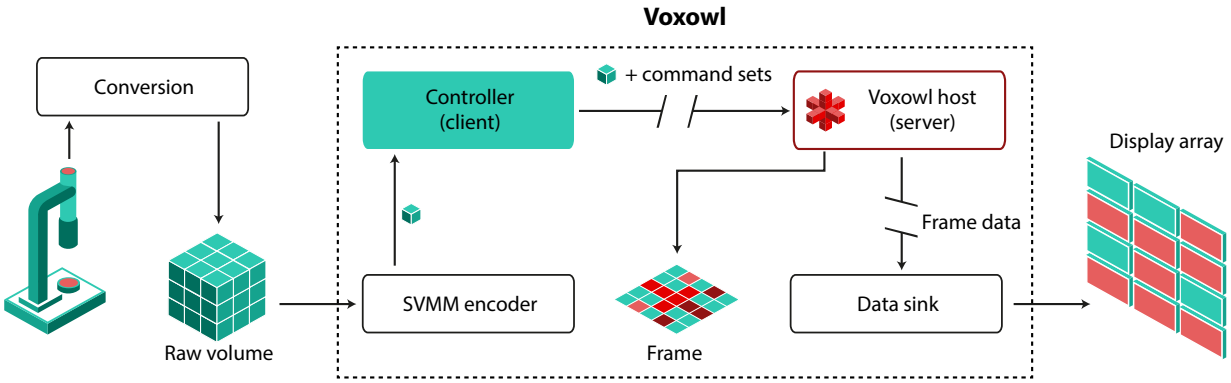


Figure 1.1: Work flow diagram

[1]The authors use the term *direct volume rendering* to distinguish from other forms of volume rendering where the data is converted to polygons first.

The LIACS *Big Eye* is a display array consisting of 12 large FHD displays, capable of showing a total resolution of 5760×4320 or a little over 24 megapixels. While the sheer size and detail of images are already interesting by themselves, the array presents many scientific possibilities in the exploration of data sets on a scale that could never be achieved on regular displays. For a visualisation of the envisioned work flow, see Figure 1.1.

1.1.1 Voxowl

Voxowl is the name of the rendering and networking project that drives the work flow. Users are to submit their volumetric data set in compressed form from their workstation (the *controller*) to the Voxowl host that runs on an external system (the *server*). On this external system, multiple GPUs can be utilized to achieved interactive framerates at high resolutions while not taxing the user's workstation. Frames that are rendered, will be transmitted over a network to the display array's computer system (the *data sink*). The latter can also serve as a secondary controller to request interactive manipulations (e.g. rotate or slice of the model) from the Voxowl host.

Sending images of this size over a network while minimizing the latency at the same time, can be very demanding in terms of bandwidth. An uncompressed native-resolution image for the Big Eye is a little over 71 MB if a standard `RGB8` format is utilized. When 30 frames/second are assumed for interactive exploration, a bandwidth of more than 17 Gbit/s is required. The resulting datastream must therefore be reduced in order to be transported over existing gigabit networks. Many video streaming solutions and low-latency protocols are already available for this purpose. Our reference implementation, however, uses a less sophisticated system that simply compresses each frame on an individual basis.

1.2 Direct volume rendering

Informally, when one expands the idea of two-dimensional imaging into the realm of 3D, another axis is simply added to the coordinate system. Where a *picture* consists of matrix of square pixels (or *picture elements*), a 'volumetric picture' or `volume` consists of equally-sized, cube-shaped[2] *voxels* or *volume elements*. In fact, most 3D-captured (e.g., tomography, MRI) data exists in this way.

Mathematically, a volume, often called a *scalar field* in that context, can be seen as a function that maps a coordinate in three-dimensional Cartesian space to a set of attributes or *channels*.

$$f\colon \{x, y, z\} \in \mathbb{N}^3 \rightarrow \{a_1, a_2, \ldots, a_n\}$$

This set of attributes could be called the *voxel format* (analogous to a pixel format) and usually consists of red, green, blue and alpha (color)channels. However, many more attributes are possible, such as *intensity* (gray-scale channel), *normals*, *tangents*, *bi-tangents* and contouring information. Voxel formats are often notated along with the size of the attributes, such as `RGBA32` (r, g, b and a channels of 8 bits each).

In the 80's and 90's, the first 3D computer games pioneered the use of voxels on consumer hardware[3]. By that time, the CGI world had already discovered the limitations of volume-based rendering and had largely moved to polygon-based rendering. The polygon-based representation of three-dimensional objects uses arbitrarily oriented triangles in (real-valued) space. Due to their continuous nature and the fact that they can be stored by a simple set of data points, polygons hold a distinct advantage over volumes, both in the storage

[2] A voxel does not have a particular shape defined. A cube is merely the form that maximizes the amount of space it takes in the volume.

[3] For example, Wolfenstein (1981), Commanche (1992) and Doom (1993) used a kind of raycasting based on a height-map. This is, however, not true volume rendering by today's standards.

as well as the image quality department. Additionally, polygons can efficiently be scanline rendered, which significantly improves speed compared to the slow raycasting used on volumes. This, and the fact that around the new millennium hardware-accelerated polygon rendering became available to consumers, made volume rendering a less favorable choice. In fact, many solutions exist where the volumetric data is converted to polygons first in order to benefit from the widespread available hardware acceleration of polygon rendering.

However, direct volume rendering has a distinct set of advantages. First of all, all attributes of the image can be encoded in a uniform way. Color information, normals and texture can all be stored directly into the voxel. This enables the use of high-precision texturing without repetition. Alpha-blending is really straightforward as no depth sorting is required. For raytracing, the ray-object intersection is greatly simplified compared to polygon-based rendering. Because a voxel can either be within the field of view or not, out-of-core rendering and culling techniques are easier to implement. Lastly, scanned 3D data, such as CT or MRI scans, can be rendered without prior conversion.

Because of these advantages, the fact that storage and computing performance have improved, and that modern GPUs (see Section 1.4) can be effectively utilized for volume rendering, volume rendering is becoming more and more commonplace. In the visual effects industry, volume rendering is not only used for effects that are inherently 'volumetric', like smoke, but also lends itself well to large scenes where large amounts of low-level manipulation is required. In gaming, completely volumetric scenes are uncommon, but voxel-based techniques are often used to enhance the performance of traditional rendering (for example, voxel-based global illumination [THGM11]). In the field of (scientific) visualisation - where volume rendering was already ubiquitous - images can now be manipulated interactively, see for example GigaVoxels [Cra11].

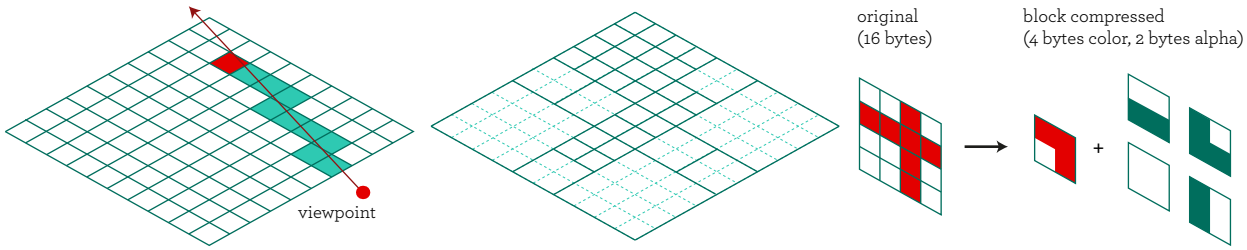
Over the years, many different direct and indirect volume rendering algorithms have been divised. For example, *Splatting* [RL00] and *Shear-Warp* [SM02] are both direct methods based on mathematical transformations. An example of indirect volume rendering is the *Marching Cubes* algorithm. Marching Cubes translates combinations of voxels into polygons which can be rendered using polygon-based renderers. Many more algorithms and variations exist and the reason for choosing one of them is usually a compromise between quality and speed. In this thesis, we focus completely on *discrete raycasting* as it offers superior image quality.

1.3 Discrete raycasting

Discrete raycasting (or simply 'raycasting') is a volume rendering algorithm that operates by traversing the dataset along imaginary rays of light. It is among the slowest algorithms, but gives an excellent image quality. Usually, a ray is cast from every pixel on the screen and is subsequently tested for intersection with the bounding box of the volume. If such intersection occurs, the ray enters the volume from the calculated entrance point. Up until here, it is very similar to *raytracing*. Raytracing operates, however, only on continuous space where as raycasting does not.

When the ray enters the volume, the volume must be traversed along the ray until either a non-transparent voxel is encountered or the volume is left on the opposite side. Traditionally, a *digital differential analysis* algorithm is used to determine the next voxel that is visited by the ray. Bresenham's line algorithm [Bre65] is perhaps one of the best known algorithms for 2D grids. For volume traversal, the '3DDDA' algorithm by Fujimoto et al. [FI85] serves as basis for many modern algorithms, including the one by Amanatides and Woo [AW87]. Figure 1.2a visualizes how the DDA algorithm traversel a grid.

As we will see later in Section 1.4, these algorithms are not directly usable in modern GPGPU computing due to heavy branching and their use of scalar operations. Many efforts exist to divise implementations that allow for efficient parallelization. In this thesis, another variation of [AW87] will be proposed that is optimized for our decompression algorithm.



(a) Ray traversing a 2D grid (b) Irregular grid of an SVO (c) Simple block compression scheme

Figure 1.2

1.4 GPGPU and Cuda

With the advent of *general purpose GPU computing* (GPGPU for short), the graphics hardware can be programmed to almost any task, including that of direct volume rendering. *Shader programs* allow specific parts of the image pipeline to be reprogrammed while leaving the rest intact. Frameworks such as OpenCL, Cg and CUDA can reprogram the *stream processors* of the GPU, allow for a much greater control and are truly 'general purpose'. Both methods, and their combinations, have been successfully used to implement direct volume renderers and discrete raycasters in particular. This research mainly focusses on CUDA to implement and test the proposed algorithms on the GPU. CUDA is NVidia's proprietary GPGPU language that is developed for recent NVidia chipsets (we will use CUDA 7.5 on Maxwell generation GPUs). We will, however, attempt to generalize as much as possible, as similar results could also be achieved with other GPGPU languages such as OpenCL [FVS11].

Raycasting lends itself well to implementation on the GPU. Most operations are inherently parallel, do not depend on each other and scale well. There are many obstacles, though. Algorithms that have traditionally been optimized for CPU, use as few floating-point operations (FLOPS) as possible (or replace them with integers), use many scalar operations, and rely on the CPU to efficiently handle branching. Modern GPUs however, are optimized for floating-point vector operations and perform poorly on branched code. This demands a radically different approach on code optimization. Another caveat is memory usage. While modern GPUs have several gigabytes of RAM, their memories are still a magnitude smaller than the CPU RAM. Moreover, because of its architecture and small caches, the GPU generally responds poorly to randomized access patterns to global memory.

1.5 Voxelmap compression schemes

While GPUs are in particular limited by their memory capacity, volume data tends to be large in general. For example a (not exceptionally large) voxelmap of 1024^3 voxels stored in 32-bit integers requires 4 Gb of memory in an uncompressed form. This size increases cubicly with the increase of resolution and it is not uncommon for a volumetric dataset to weigh several gigabytes up to many terabytes. The naive approach would be to store this data inside a large array. Such an array will be called a *voxelmap* throughout this thesis. While this method benefits from favorable constant-time access inherent to arrays, it severely limits the usable size of a dataset. However, many effective methods of compression and encoding schemes have been developed and some of the most noteworthy will be reviewed in the following subsections.

1.5.1 3D textures

While not strictly a compression scheme, the use of the GPU's texture units give a substantial improvement in performance for direct volume rendering. Recent versions of CUDA, for example, support 3D textures with resolutions up to 2048^3 and up to 128 bits of data per *texel*[4]. Caching of these 3D textures is optimized for spatial access patterns, as opposed to regular *linear memory*. This alleviates some of the problems concerning locality of ray traversal. Consider the following formula to translate a coordinate on a voxelmap ($u \in \{0, \ldots, U-1\}, v \in \{0, \ldots, V-1\}, w \in \{0, \ldots, W-1\}$) to a linear memory position n. (Assuming column-major ordering)

$$n = wUV + vU + u$$

As shown in Figure 1.2a, ray traversal (such a 3DDDA) always accesses an adjacent voxel each step. However, to obtain for example voxel $(u, v, w+1)$, a large jump in linear memory is needed to $n + UV$. That will be well outside any regular cache for practical sizes of U and V. Thus for every ray cast step, a complete access to global memory is required, while wasting the entire cache at the same time (a voxel is never revisited). The hardware texture units partially solve this problem by caching 'blocks' instead of linear strips of data. This and the fact that they also provide automatic padding of data to optimize the efficiency of read operations, makes 3D textures widely used for volumetric data.

1.5.2 Sparse Voxel Octree

Probably the most widely used datastructure for volumetric data, is the *Sparse Voxel Octree* (SVO) [LK10]. An octree is a tree where each node is cube-shaped and consists of 2^3 child-pointers. An SVO is created by continuously subdividing the original volume by two in each direction. Thus each next level in this tree, represents an eight times larger resolution, with the bottom-level containing the original resolution and the root being a single 2^3 cube. This structure can easily be compressed by iteratively removing the leaves that contain eight identical voxels. Large continuous regions are then described by a larger, top-level node, while small details are contained within the lower regions of the tree - hence the classification 'sparse'. See Figure 1.2b.

SVOs generally give a good compression ratio at the cost of more complexity. Naturally, given a volume of size N^3, the worst-case complexity for accessing a single voxel is $log_2(N)$ while the average-case depends heavily on the input. To alleviate some of this increased complexity, SVOs can be used to efficiently skip empty regions by modifying the raycast algorithm. Moreover, because successive accesses are likely to have a large degree of spatial locality, the traversal of the tree can be further optimized by the use of a stack to remember parent nodes. Unfortunately, GPUs respond poorly to these optimizations due to the use of complex data structures such as a stack.

1.5.3 Run-length encoding

Run-length encoding (RLE) is another frequently used algorithm. For the sake of brevity, the principle of RLE is assumed to be known. Being a one-dimensional encoding, RLE can only be used on one axis of the voxelmap. The result is a two-dimensional grid of irregularly sized 'columns' of RLE data. RLE can be used to achieve favorable compression ratios [FO11] while, like SVOs, it substantially increases the number of 'steps' it takes to reach a single voxel. Because compression really occurs over one axis, this complexity becomes somewhat dependent on the angle of view. RLE therefore lends itself especially well for data that already has an increased density in a certain direction, such as height maps.

[4] Analogous to a pixel in an image, the term *texel* refers to one element of a texture

1.5.4 Transformations

Mathematical transformations can be used to express the values of a dataset into a different domain (frequency domain is used in particular in this context). While these transformations do not compress the data as such, the resulting (transformed) data may be easier to compress with, for example, run length encoding. Moreover, a lossy compression may be achieved efficiently by leaving out the high-frequency components of the (transformed) data at a minimal degradation of the image quality. Mostly the *discrete cosine transformation* [Wat94] and the *wavelet transformation* [wav99] have been utilized to compress image data[5].

1.5.5 MIP maps

MIP maps (MIP standing for *multum in parvo*) is a technique that originates from the rendering of 2D textures. It works by recursively dividing the resolution in half and storing each subsequent downscaled image alongside the original. Each separate image is called a *mipmap level*. In its original context, MIP mapping is used to enhance the filtering quality of the texture, however, in volume rendering it can be used to limit the ray traversal based on level-of-detail (LOD). For example, if parts of the image are really far away from the viewer, one of the coarser mipmap levels can be used, resulting in fewer steps when traversing the volume. In a similar fashion, inherently *anti-aliased* images can be generated (*ray tracing with cones*) [Ama84]. For volumes, each subsequent mipmap level is only $(\frac{1}{2})^3 = \frac{1}{8}$ as large as the previous level. Expanding this series gives:

$$\sum_{i=1}^{\infty} \frac{1}{8^i} = \frac{1}{7}$$

This shows that a full MIP map expansion adds only about 14,3% to the size of the voxelmap. However, MIP maps are often combined with SVOs in this context, making the increase in size due to MIP mapping negligible.

1.5.6 Block compression

In the literature only few applications of *block compression* in the field of volume rendering are described. However, this technique has many potential benefits in this area. Block compression is an umbrella term for several techniques that compress data per block of a fixed size, while producing an output with a fixed rate. The biggest advantage is that no datastructure is necessary, as regularly formatted data remains regular. This property makes it very well suited to (3D) texture compression. Disadvantages are the limited compression ratios and the inherent loss of information; block compression is a 'lossy' method by definition.

A famous example of block compression is S3TC (S3 Texture Compression) [S3][6]. S3TC comes in the forms of DXT1 to DXT5, all providing similar mathematical schemes of encoding a 4×4 block into fixed-width integers with ratios from 4:1 to 6:1.
In video compression, color is often separated into *luminance* (brightness) and *chroma* (color) channels. Owing to the fact that human vision is more sensitive to change in brightness than in color, *chroma-luma* compression reduces the resolution of the chroma channel without much image degradation. Video formats often denote the applied chroma-luma compression as a ratio such as '4:2:2' or '4:2:0', giving the ratio of luminance information, horizontal-, and vertical chroma information. Figure 1.2c shows a very simple block compression

[5] JPEG and JPEG-2000 are good examples of these
[6] S3TC is currently covered by a patent that expires in 2017.

scheme inspired by '4:2:2' chroma-luma compression. The luma part is replace by a single-bit alpha channel and color is simply stored as a RGBA quadruple. We will use this scheme later in our experiments.

1.6 Problems and questions

There are many lossy compression algorithms for 2D images - both static and motion - but very few exist for volumetric data. Most existing algorithms compress their data loss-less and for the purpose of rendering only, while storage on disk and exchange over network could also greatly benefit from this compression. We also believe that loss in quality, to a degree, can be acceptable if the gain in storage requirements is large enough. As presented earlier, direct access to an array-based volume is fast but requires a significant amount out memory, while an octree-based solution sacrifices the constant-time access for a smaller memory footprint.

Our goal is to develop an improved compression scheme that could potentially be used as a means of data exchange as well as directly and without modifications during rendering. To serve as a framework and as a means of evaluations, we pose the following questions throughout this thesis:

- Can we devise a *lossy* compression algorithm that is useful for practical volumetric data sets?
- Can we improve upon existing methods, namely sparse voxel octrees.
- By which factors are rendering performance, compression ratio and image quality affected? Can we find an 'optimal' balance?
- Using compression, can real-time visualisation with a large voxelmap be achieved on a single GPU? [7]

In addition to these questions, we hypothesize that the use of block compression will improve the size-quality ratio. We also believe that by reducing the amount of mipmap levels - or the depth of the tree in SVO-terms - the performance will improve proportionally. We will use our test result to verify these theories.

1.7 Sparse Voxel MIP maps

In an attempt to overcome the problems mentioned earlier, a novel compression data structure is proposed, that is based on SVOs, volume MIP mapping and block compression. These *Sparse Voxel MIP Maps* or SVMMs are designed to efficiently make use of GPU's 3D texture memory while also limiting the amount of mipmap levels. In order to achieve compression in combination with MIP mapping, we divide the mipmap levels into individual blocks of voxels. These blocks can have an arbitrary size and can be 'left out' (hence *sparse*) when their entropy falls below some threshold. While we do not consider SVMM to be a tree per se, a meta-structure is intervowen into the MIP map data that much resembles an N^3-tree. It is using this structure that we can localize blocks in a sparsely populated mipmap level.

Some of the key-features of SVMM are: support for multiple voxel formats, efficient storage to disk and a variable lossy compression. It could potentially be expanded with out-of-core rendering (see Section 5.4.1). Moreover, we make use of block compression to further reduce size and improve the size-quality ratio.

1.8 Related Work

Much research has been done into the efficient storage and level-of-detail representation of volumetric data. The use of RLE can be found in the work of Forstmann et al. [FO11] and also in conjunction with wavelet-

[7] As far as the last question is concerned, we consider an RGBA voxelmap of 2048^3 - a little over 36 Gb - to be 'large' at the time of writing.

compression by Kim et al. [KS99]. We will limit ourselves to SVO- and/or MIP Map based algorithms as these most directly relate to this research. For a an overview of contemporary volume compression schemes, we direct the reader to the work by Rodrguez et al. [RGG+14].

The work of Laine et al. [LK10] describes the efficient use of SVOs for GPU rendering. Their research can be considered state of the art regarding Sparse Voxel Octrees. While their approach is very general, the decision is made to focus mainly on the surface information of the data sets. Moreover, their representation greatly benefits from *contouring information* which is only available if the original data is a mesh instead of a volume.

GigaVoxels by Crassin [Cra11] appears to be a very mature project offering an all-round GPU based solution. Their efforts also include a MIP Map - SVO hybrid approach that uses a *kd-start* algorithm to traverse the octree without stack and an efficient 3D texture look-up to fetch the individual voxel data. This data representation allows for *ray tracing with cones* [Ama84] which provides clever anti-aliasing. Furthermore, out-of-core rendering (streaming) is utilized to allow very large data sets to be rendered on a GPU with limited memory capacity.

The work by Gobbetti et al. [GMG08] is in many ways similar to that of Crassin [Cra11]. It focussing primarily on out-of-core rendering, while the work of Crassin [Cra11] discusses more facets of image quality, such as shadowing and anti-aliasing. They also implement a stack-less ray traversal algorithm, but they use a special neighbour indexing structure to achieve this.

As far as the rendering of the compressed data - in this context by means of discrete raycasting - is concerned, the method of traversal is of great importance. It has been shown that the traversal of the volmetric data is the largest factor in throughput for direct volume rendering [Lev90]. Therefore, it has been the subject of many studies. The original 3DDDA algorithm by Fujimoto et al. [FI85] and the subsequent improvement by Amanatides et al. [AW87] are not suited for modern GPGPU implementation. Many efforts to optimize these algorithms for *stream processing* have been made, for example the work by Es et al. [E07].

1.9 Contributions

In this thesis, we will discuss the theory of the proposed SVMM datastructure as well as the conceptual implementation of its encoder and decoder. We will specifically focus on optimizing the raycaster and decoder for implementation using CUDA. Since the use of GPGPU programming is now so commonplace in volumetric rendering, we believe it to be a realistic testing environment. We will also introduce the reader to our methods of evaluation and present its results. In the evaluation of our compression method, we will assess both rendering performance and compression ratio.

The remainder of this thesis is structured as follows:

- Chapter 2 will introduce the SVMM datastructure and describe its encoding
- Chapter 3 will discus challenges encountered during decoding and rendering. It will also put the decoding in perspective by briefly touching upon the entire rendering pipeline and its potential problems.
- In Chapter 4, the method of testing and its results will be elaborated on.
- In Chapter 5 we will discuss and conclude our contribution.

Chapter 2

Datastructure for storage of compressed volumes

2.1 Theory

In essence, the SVMM of a volume is generated by recursively creating copies of a lower resolution and storing them alongside each other, just like a regular MIP map. The resulting chunks of data are called the *mipmap levels*, where level 0 or the *base level* is the original, non-scaled volume. Where a regular mipmap level divides each side of the volume by two, SVMM can divide by an arbitrary number; this is controlled by parameter called `blockwidth` or w for short. The voxel (x, y, z) at level $n + 1$ is now calculated by averaging a w^3 region at level n starting at (wx, wy, wz). This region gives us the definition of a *block* on level n. Subsequently, a *equivalent voxel* is defined as being the single voxel on level $n + 1$ that corresponds to this block on level n. Thus an equivalent voxel always contains the average value of all the individual voxels in the block it corresponds with. This subdivision is repeated recursively until the resulting dimensions of a level are smaller than or equal to a certain threshold. This is parameter r or the `rootwidth` and the last level is subsequently called the *root level*. The root level is now the lowest resolution mipmap level generated.

The values of w and r can be adapted to the size of the given volume and need to be balanced between the level of compression and the depth of the tree. Small values for w would create a potentially very deep tree (note that $w = 2$ generates an octree) while large values of w would give almost no compression. The value for r should be as large as possible as it eliminates the (needless) higher levels in the mipmap structure. However, the entire root level should still fit conveniently in memory, as we will later see. Moreover, a large r obviously creates a very shallow tree and (almost) no compression can be applied.

When compared to an SVO, there are many similarities. An SVO can, for example, also be seen as an elaborate MIP Map structure. However, the fact that the SVMM datatstructure has block widths > 2 and these widths can be selected per-level, makes it more like a N^3 tree rather than an octree. We deliberately avoid the term tree though, as an SVMM is conceptually closer to a collection of self-contained, multi-resolution volumes. Figure 2.1 gives a complete overview of the datastructure.

2.1.1 Compression

The procedure above only generates a MIP map, but does not compress the volume just yet (in fact, it adds overhead). Therefore, an extra step is added when the average of a block is calculated. This step is designed

to determine whether a certain block contributes significant information to the next mipmap level and if it does not, one could choose not to store this block. If a block is not stored, its equivalent voxel is called *terminal*, i.e. no deeper mipmap levels follow. Observe that all voxels at level 0 are terminal. The resulting tree now becomes unbalanced (i.e. not all the leafs are on the same level) much like with SVOs. When the difference between each individual voxel of a block and its equivalent voxel (the average) is smaller that a certain parameter δ, it is said to be *homogeneous*. If all voxels inside a homogeneous block are terminal, it may be discarded and its corresponding equivalent voxel becomes terminal. As a result, empty blocks or blocks consisting of equal voxels are not stored.

By changing δ, the amount of lossy compression that is applied can be influenced. Just like w and r, the parameter δ can be tuned to achieve a balance between performance (image quality) and storage requirements. A large value for δ will throw out many blocks whose individual voxels are replaced by their average. This would theoretically result in a smaller memory footprint as well as a shorter rendering time (less blocks mean less traversal). The resulting volume will exhibit a 'blocky' appearance much like a JPEG-compressed image. The influence a certain value of δ has, depends on the number of attributes n per voxel. Therefore, a certain intended value of δ needs to be weighted by the square root of n. This way, the same value of δ influences the inequality with different amounts of attributes in a similar way. In order to handle this transparently and to improve readability, we introduce the *quality factor* q that ranges for 0 to 100. The quality factor directly determines δ by the equality $\delta = 1 - \sqrt{n}\frac{q}{100}$.

Lastly, there is the option of applying additional lossy, fixed-rate (block) compression to the level-0 blocks separately. This gives an additional compression without adding complexity, and the compressed blocks can easily be stored using hardware 3D textures. The simplest method is to only store a one-bit alpha value for each voxel and discard the color information (see Figure 1.2c). This is a bit like the chroma-luma compression commonly utilized in video. It retains all the detail of the given block, while the color information is extrapolated from its corresponding equivalent voxel. Naturally, this will degrade image quality and depending on the original volume this may or may not be acceptable. However, in it simplest form this may already give a substantial 32:1 compression ratio on the mipmap level 0 without loss of detail (assuming 32 bits are used to store one regular voxel). Of course, less aggressive block compression methods can be employed. Simply converting the base-level to grayscale yields a 4:1 gain, while using a more

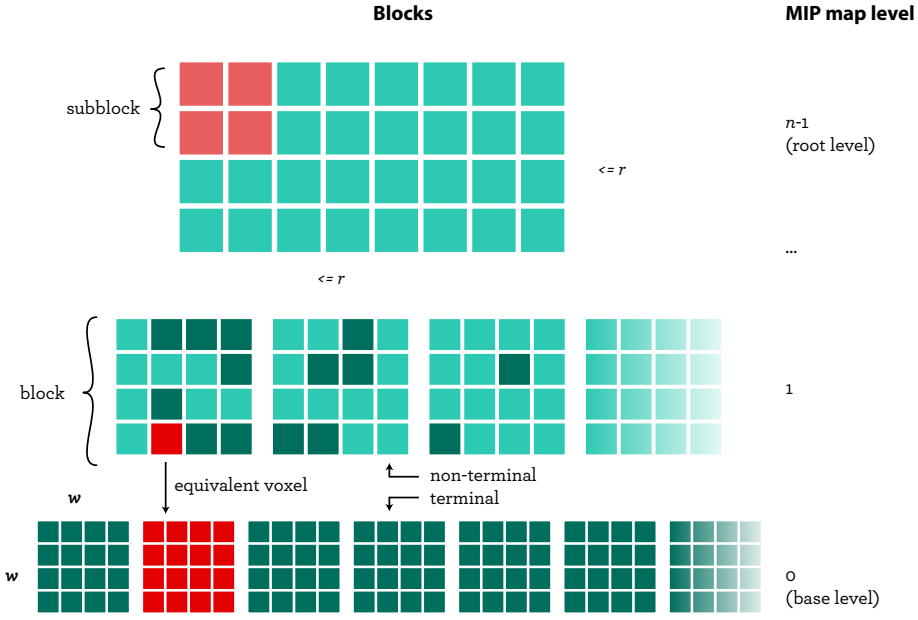


Figure 2.1: Sparse Voxel MIP Map

sophisticated algorithm such as S3TC could be used to achieve better image quality.

2.2 Implementation

Thanks to the simplicity similar to regular MIP Maps, SVMM encoding a volume can be highly efficient. The algorithm consists of two fundamental procedures: `EncodeLevel` and `EncodeBlock`, encoding a level and a single block respectively. Levels are encoded recursively until the remaining width of the current level is smaller or equal to r. Blocks are stored as *voxelmaps* that can have different single- and multi byte voxel formats and can be stored as 3D textures by the GPU decoder. To encode structural information, a simple header precedes each level. Furthermore, the concept of *sub blocks* is introduced to efficiently store offset information on a per-voxel basis. The next section will discus these concepts in detail. They are also implemented in Voxowl as a reference for comparison.

2.2.1 Voxelmap formats

The format of the voxels can be defined on a per-level basis. For example, all non-base levels can be set to use a format that stores full color information, while the base level only stores bitmaps containing a 1-bit alpha channel for each voxel. For the base-level this may be any conceivable format (including for example S3TC), while the other levels are restricted to 32 bits or 16 bits per voxel. These special formats, preserve the 5 least significant bits for structural information about each voxel, as can also be seen in Table 2.1. As as result, `RGBA8` formats lose 5 bits from their alpha channel. It is our experience, however, that a true 8-bit alpha channel is rarely needed. Reducing the alpha to 1 bit, on the other hand, has been shown to result in round-off errors during compression. Moreover, if a true semi-transparency is needed, a different voxel format may be devised (for example, `RGBA7776`) or the `DENSITY8` format can be used (at expense of color information).

2.2.2 Subblocks

To store a block's offset at level $n-1$, its equivalent voxel at level n reserves its 4 least-significant bits. This could, obviously, only address 32 blocks at most. By logically dividing each block into *subblocks* of 2^3 and combining the 4 offset bits of each voxel by bitshifting, a total of 32 bits can be stored in one subblock. It only makes sense to divide non-base levels into subblocks, hence the limitation on the number bits per voxel for these levels. Each subblock thus consists of 2^3 equivalent voxels that correspond to some block on a lower level. This means that, when all these voxels are non-terminal, we should be able to locate 2^3 different blocks on this lower level. Using the 32 bits of the subblock collectively, exactly one block can be located by means of an offset. The (potentially) other $2^3 - 1$ blocks can now be located by counting the number of terminal voxels in the subblock, as can also be seen in Figure 2.2.

The voxels within the subblock s on level n are addressed in a column-major manner as $s[i]$, where the collective offset points to the block that corresponds the first non-terminal equivalent voxel in the subblock

bits	31-16	15-8	7 - 5	4	3 - 0
32 bits formats	Color (24 bit)		Alpha	Terminal-flag	Part of block group offset
16 bits formats		Intensity (8 bit)	Reserved	Terminal-flag	Part of block group offset

Table 2.1: Voxel formats for non-base levels

(which is not necessarily $s[0]$) on level $n-1$. The offset of this block can now be calculated by:

$$\texttt{offset}(s) + \sum_{k=0}^{i-1} \begin{cases} 1, & if\ \texttt{terminal}(s) \\ 0, & otherwise \end{cases}$$

With the use of subblocks, a maximum of $2^{32} * w^3$ individual voxels can be stored per mipmap level. Given a conservative compression rate of 15% and a w of 2, the minimum largest volume size would be roughly 21600^3. Because of the spatial caching the GPU 3D textures provide, accessing a single voxel is still reasonably efficient. During traversal, however, the access pattern is not random and many voxels within the same block are fetched, further diminishing the overhead causes by the distributed storage of offsets.

2.2.3 Encoding a level

Encoding can be described top-down: from the entire process, to a per-level basis and finally to the block level. See Figure 2.3 for a block diagram of the simplified algorithm. The input for the encoder is a voxelmap V_0 and its output a set of mipmap levels $\{L_0, L_1, \ldots, L_{n-1}\}$. The user specifies values for w, r and δ. V_0 is the original volume and has a *size* and a *type*. The first level that is encoded is therefore the base level L_0. The subroutine `EncodeLevel` takes a voxelmap V_i and returns a sequence of blocks that make up the level L_i, and a new voxelmap V_{i+1}. V_{i+1} is a *temporary* voxelmap that is w times smaller along each axis than the input V_i. V_{i+1} is also w^{i+1} times smaller than V_0, this number is the *mipmap factor* and is stored in the level's header. $V_0 \ldots V_{n-1}$ are encoded recursively until the width of a certain V_{n-1} is equal or smaller than r. This means that there are only L_{n-2} levels generated. The root level, L_{n-1}, is simply a level consisting of one block, namely V_{n-1}.

2.2.4 Encoding a block

A level L_i is a sequence of blocks $< B_0, \ldots, B_{K-1} >$, while a block itself is again a voxelmap of a certain type. Before blocks can be written, the input voxelmap V_i must be a multiple of w and a multiple of 2. Necessary padding will be added. V_i will then be divided into blocks $< B_0, \ldots, B_{L-1} >$ which are mapped onto the sequence of blocks $< B'_O, \ldots, B'_{K-1} >$. Note that K and L may or may not be equal. The subroutine

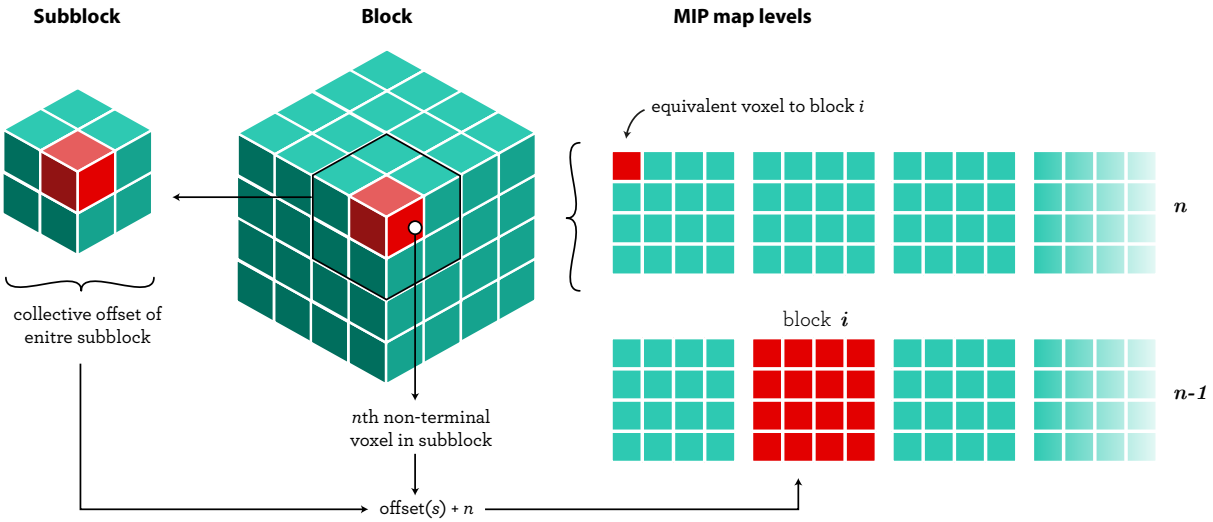


Figure 2.2: Subblock used to store offsets

`EncodeBlock` takes a certain input block B_l and an output block B'_k. The average $\boldsymbol{\alpha}$ of all voxels in B_l is computed. If B_l is *not* homogeneous, , it is converted, compressed and written to B'_k.

Both compression steps (the δ-parameter and block compression) are applied during the `EncodeBlock` step. First of all, the δ-parameter is applied during the test for homogeneity. We have empirically determined that using the *vector distance* $|\boldsymbol{\alpha} - \mathbf{v}_i|$ between the block average $\boldsymbol{\alpha}$ and each individual voxel $\mathbf{v}_i$ in the block, is a good measure for homogeneity. Both the average and the individual voxel are therefore defined as vectors of the voxel's attributes $< a_0, a_1, \ldots, a_{n-1}$. Homogeneity is now defined as follows:

$$\mathbf{v}_i = B_l[i]$$

$$\text{homogeneous}(B_l) = \forall_j : 0 \leq j < size(B_l) : \sqrt{(\alpha[0] - v_j[0])^2 + \cdots + (\alpha[n-1] - v_j[n-1])^2} \leq \delta \wedge \text{terminal}(v_j)$$

For non-base levels, in order to produce correct subblocks, the conversion of blocks as described above, is performed in groups of 2^3. Starting at block B_l, the resulting set of 8 averages $\{\alpha_l, \ldots, \alpha_{l+7}\}$ become the equivalent voxels that correspond to the blocks of the sequence $B'_k, \ldots, B'_{k+7}$ that are not homogeneous. In order to do so, these averages will make up the subblock in V_{i+1} starting at $V_{i+1}[l]$. Furthermore, the offset o of the first non-homogeneous block of the sequence $B'_k, \ldots, B'_{k+7}$ must be determined (if any). The individual voxels $\{v_l, \ldots, v_{l+7}\}$ from the subblock $V_{i+1}[l]$ will then be set according to ($< m - n >$ denoting an assignment to selective bits):

$$\forall_i : 0 \leq i < 8 :$$

$$v_{l+i} < 31 - 8 > \;:=\; \alpha_{l+i} < 31 - 8 > \qquad \text{Assign color}$$

$$v_{l+i} < 7 - 5 > \;:=\; \alpha_{l+i} < 7 - 1 > \,/\, 255 \times 8 \qquad \text{Assign alpha}$$

$$v_{l+i} < 3 - 0 > \;:=\; o \;\&\; \texttt{0xf} << 6i \qquad \text{Assign a 4 bit part of the offset}$$

$$\text{terminal}(v_{l+i}) \;\; \textit{iff} \;\; \text{homogenous}(B_{l+i}) \qquad \text{Set terminal equiv. voxels}$$

For the base level, no terminal flags need to be set. However, block compression can optionally be applied to the non-homogeneous blocks.

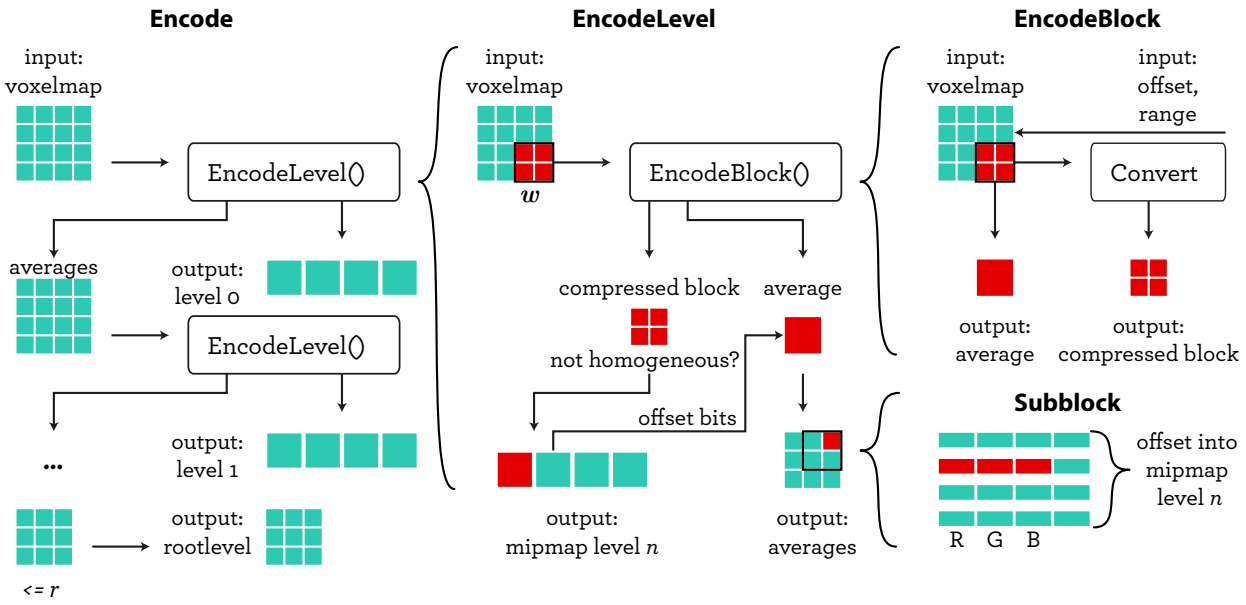


Figure 2.3: Block diagram of the encoder

Chapter 3

Decoding and Rendering

In this chapter, a method of decoding SVMMs will be presented as well as its incorporation to a modified version of the algorithm by Amanatides et al. [Ama84]. Furthermore, some of the specific challenges related to rendering of SVMMs and volumes in general will be dicussed.

3.1 Decoding principles

From the most simple and naive perspective, an SVMM compressed volume could be treated as a black box that 'wraps around' a voxelmap. A layer could be created that acts as a translator between the SVMM encoded volume and this *logical* voxelmap. To this end, one function $f\colon \{x, y, z\} \in \mathbb{N}^3 \rightarrow \{a_1, a_2, \ldots, a_n\}$ would be needed that would abstract away all internals and map a logical coordinate (x, y, z) within the volume to its set of attibutes. Internally, we would have to traverse the correct blocks beginning at the root of the SVMM. Given a *logical* index $\mathbf{v}$ within the volume, we need the corresponding *absolute index* within the root level n. This level's mipmap factor p_n is used in

$$abs_n(\mathbf{v}) = \lfloor \frac{\mathbf{v}}{p_n} \rfloor$$

Since the root level always consists of precisely one block, the absolute index gives the index of interest $\mathbf{v}'_n$, corresponding to logical $\mathbf{v}$ in the volume, directly. If at anytime holds that $terminal(L_n[\mathbf{v}'])$, we can stop and return the attributes (color) at that particular index.

If a node at a particular index is not terminal and there are more levels available, the resulting voxel must be *refined*. When a voxel is refined, a node in a lower mipmap level is visited of a finer resolution and thus gives a more specific result (hence refinement). The first step is to calculate the offset into this level $n-1$ to find the corresponding block to $\mathbf{v}'_n$. For this, however, we need the entire subblock that encompasses $\mathbf{v}'_n$. To find the subblock's index, we can simply round all components of $\mathbf{v}'_n$ down to the nearest multiple of two (a subblock is always 2^3) to obtain this index $\mathbf{s}$. By the method described in Section 2.2.2 we can now find this $o = offset_{n-1}(\mathbf{v}'_n)$.

Having entered a lower mipmap level $n-1$, the absolute index must again be calculated. Being a non-root level (multiple blocks), this now means that $abs_n(\mathbf{v}) \neq \mathbf{v}'_{n-1}$. In order to determine $\mathbf{v}n-1'$, the *relative index* into the block B must be calculated. The previously calculated offset o can now be used to obtain B, while

the blockwidth w_n is used to calculate the relative index:

$$B = L_{n-1} + o \qquad L_{n-1} \text{ being the begin address of level n-1}$$
$$\mathbf{v}'_{n-1} = rel_{n-1}(\mathbf{v}) = abs_{n-1}(\mathbf{v}) \bmod w_{n-1}$$

Having calculated $\mathbf{v}'_{n-1}$, indexing the block as $B[\mathbf{v}'_{n-1}]$ now gives the attributes mapped to the logical $\mathbf{v}$ at level $n-1$. This process of refinement is repeated until a voxel is terminal or it is at the base mipmap level.

3.1.1 Obtaining multiple voxels

The method outlined above is not usable in practice, as it involves lots of redundant computations. Moreover, the refinement step relies on many integer divisions and modulus operations that are known to be extremely slow. The latter is solved by demanding that any w is a power of two. Every p is then automatically also a power of two and many division operations can be optimized away by using bit shifts. Another fortunate circumstance is that in the practical cases, the voxels that need to be extracted are in close vicinity. This holds for ray traversal, as we will see later, but also for the case that the entire data set needs to be decoded back to a voxelmap. By using a regular stack-based depth-first traversal, every terminal block (leaf) can be efficiently visited owing to the fact that the SVMM is essentially a tree.

3.2 An algorithm for ray traversal

The 3DDDA algorithm and the algorithm explained by Amanatides et al. [Ama84] are quite simple and compact. While they could be used in conjunction with the method outlined in Section 3.1, the performance would be extremely poor due to multiple reasons. In the following sections we will outline these problems and our proposed method to overcome them.

3.2.1 Baseline algorithm

Before the problems are to be discussed, we give an optimized version of the traditional raycast traversal algorithm as a baseline. This algorithm is also implemented in our experimentation code as basis for comparison using non-compressed data sets.

The traversal begins with an origin $\mathbf{o}$ and a ray direction $\bar{\mathbf{d}}$. These variables are obtained from the position of the camera and the screen-space coordinates of the fragment, respectively. In order to be able to step through an arbitrary volume that is arbitrarily oriented in space, we transform all coordinates to the volume's model-space. Since, for example, the origin and ray vector are in world-space, we multiply them by the inverse model-view-matrix. Regarding the volume itself, we assume that we can represent the volume by an Axis Aligned Bounding Box (AABB) of unit size with its center positioned at $(0,0,0)$. Therefore, for a volume of N^3, each voxel is a cube whose width, height and depth is $\frac{1}{N}$. If the volume is not cubical, the largest side is set to unit length while the other sides are a fraction to their proportions. By performing a ray-AABB intersection test (described by Glassner et al. [ea89] and by Williams et. al [WBMS05]) as can be found here [Bar11], the coordinate of entrance $\mathbf{a}$ is obtained (assuming the ray hit the box). Using a and $\bar{\mathbf{d}}$, Algorithm 1 now shows the process of stepping through the volume.

The main operation of the algorithm is to find the axis with the smallest distance to the next dividing plane and increment the index in that direction. The original algorithm by Amanatides et al. [Ama84] uses heavy branching to determine which axis to choose. The major difference is that we vectorize the branching code

Algorithm 1 Baseline ray traversal

Require: Volume V of size $\mathbf{S}$.
Require: Ray entrance point $\mathbf{a}$, bounding box b and ray direction $\bar{\mathbf{d}}$.

1: $\ell := (max)(S.x, (max)(s.y, s.z))$ ▷ Length of the largest direction of the volume's size
2: $\mathbf{v} := (\mathbf{a} + b.\mathbf{max}) * \ell$ ▷ Index where ray hits within the volume
3: $\Delta := (\frac{dx}{d}\bar{\mathbf{d}}, \frac{dy}{d}\bar{\mathbf{d}}, \frac{dz}{d}\bar{\mathbf{d}})$ ▷ Slope of $\bar{\mathbf{d}}$ within the volume
4: $\mathbf{t_{max}} := \text{sign}(\bar{\mathbf{d}}) * (\lfloor \mathbf{v} \rfloor - \mathbf{v}) + (\frac{1}{2}\text{sign}(\bar{\mathbf{d}}) + \frac{1}{2}) * \Delta$ ▷ Relative distance to each of the dividing planes
5:
6: **while** true **do**
7: $c := \texttt{tex3D}(V, \lfloor \mathbf{v} \rfloor)$
8: $\texttt{frag.color} := \texttt{blendF2B}(c, \texttt{frag.color})$
9:
10: **if** $\texttt{frag.color}.a < \epsilon$ **then return** ▷ Stop if the transparency falls below some threshold
11: **end if**
12:
13: $\mathbf{b_0} := (\mathbf{t_{max}}.x < \mathbf{t_{max}}.y, \mathbf{t_{max}}.y < \mathbf{t_{max}}.z, \mathbf{t_{max}}.z < \mathbf{t_{max}}.x)$
14: $\mathbf{b_1} := (\mathbf{t_{max}}.x \leq \mathbf{t_{max}}.z, \mathbf{t_{max}}.y \leq \mathbf{t_{max}}.x, \mathbf{t_{max}}.y \leq \mathbf{t_{max}}.x)$
15: $\mathbf{mask} := (\mathbf{b_0}.x \wedge \mathbf{b_1}.x, \mathbf{b_0}.y \wedge \mathbf{b_1}.y, \mathbf{b_0}.z \wedge \mathbf{b_1}.z)$
16:
17: $\mathbf{t_{max}} := \mathbf{t_{max}} + \mathbf{\Delta} * \mathbf{mask}$
18: $\mathbf{v} := \mathbf{v} + \text{sign}(\bar{\mathbf{d}}) * \mathbf{mask}$
19: **end while**

here, leading to the compact code on lines 13-18. On lines 13-14, all three components of $\mathbf{t_{max}}$ are compared systematically such that the vector **mask** is assigned a 1 on the smallest axis and 0's on the other two. This mask is then used to selectively update $\mathbf{t_{max}}$ and the index $\mathbf{v}$. For a more detailed review of the principles behind the algorithm, we refer the reader to the aforementioned work by Amanatides et al. and to [Bre65] and [FI85].

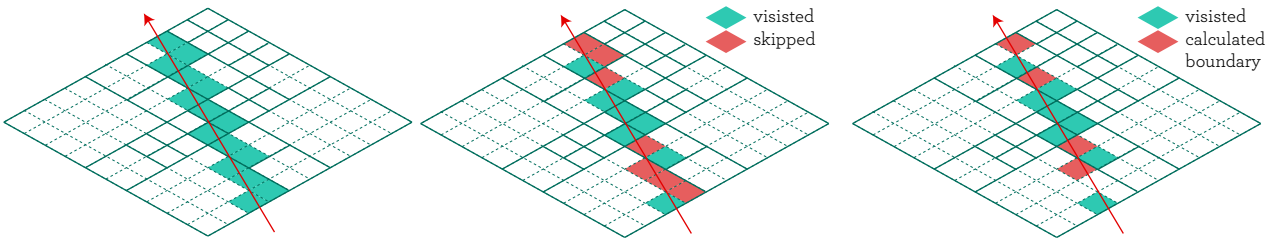
3.2.2 Depth traversal

In Section 3.1.1 a stack-based traversal method was mentioned to visit and refine adjacent voxels. While this method certainly reduces the amount of computations needed for ray traversal, a stack-based approach is not feasible. The stack datastructure uses global memory specific to one thread, which scales poorly on the GPU. Our solution is to simply re-refine each subsequent voxel instead of storing a stack of visited nodes. This may sound redundant - which it is - but it is also actually faster than using a stack on the GPU. Using a number of optimizations, however, we can severely limit the number of full traversals.

The first and most obvious optimization is to continue to traverse a block as long as the logical index $\mathbf{v}$ falls within and the voxels are terminal. Thus, the larger the value for w is, the more voxels can potentially be visited without traversing the tree again. We achieve this by storing the mipmap factor of the equivalent voxel in order to know how one step on a certain level changes the logical index. Comparing Figure 3.1a and Figure 3.1b, we can see the potential gain this provides

3.2.3 Skipping of non-base level voxels

The method mentioned in the last subsection, requires the algorithm to step through many logical voxels in order to finally reach its (useful) neighbour. Although no expensive instructions are required, it would be even better to have the ability to calculate the exact distance one has to travel to the neighbouring voxel. The problem is, however, that we can only perform a step at the smallest level (the logical voxelmap). So in the worst case, as many steps as a level's mipmap factor are required.



(a) Naive method visits all logical voxels redundantly

(b) Making all steps, only visiting relevant voxels

(c) Calculating boundaries to do at most two steps per voxel

Figure 3.1: Different methods of skipping terminal blocks

In the work by Sung [Sun91] an algorithm is proposed that calculates the boundary to the neighbouring voxel, at an arbitrary level and in a constant amount of instructions. This means we can use one 'complex step' as described by Sung (we are at the boundary then) and then one regular step to arrive at a neighbour. Figure 3.1c demonstrates this. While this is a very useful algorithm, our experiments demonstrated that the gain over the 'obvious method' is generally small. This is partly due to the fact that it uses many additional registers on the GPU, and partly due to the fact that we already reduce the amount of mipmap levels significantly, so there was little gain to be had in the first place.

3.2.4 Level-Of-Detail and Aliasing

Sometimes it may not be necessary to refine until a terminal voxel is reached. Especially when the size of a fragment becomes larger than the size of an individual voxel, further refinement actually degrades the quality of the final image. In terms of signal processing it can be said than the detail of the geometry exceeds the Nyquist-frequency of the carrier (the screen). In this case aliasing of the signal will occur, resulting in unpleasingly looking edges in high-frequent details. Additionally, due to the grid structures of both the screen and the voxels, the *moiré* phenomenon can occur. Moiré introduces false patterns where two grids of similar frequency are projected onto each other, also resulting in displeasing imagery.

The 'perfect' solution to these problems is the *oversampling* of the signal. In computer graphics, this is often called *supersampling anti-aliasing* [Cro77] and basically means that for each fragment, multiple samples are taken and then averaged. So for each pixel, multiple rays need to be cast, resulting in significantly poorer performance.

The work by Amanatides [Ama84] demonstrates a much more elegant way that can be used with raytracing. Amanatides proposes that a ray is not a straight line, but should rather be modelled like a cone. As demonstrated by Figure 3.2, we are interested in the integral of the cone instead of a singular sample hit by

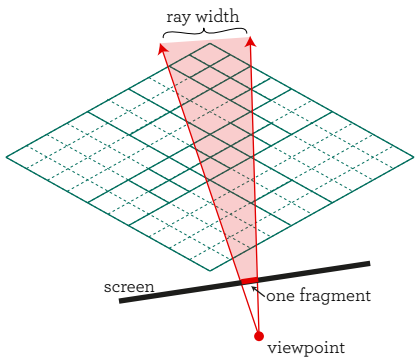
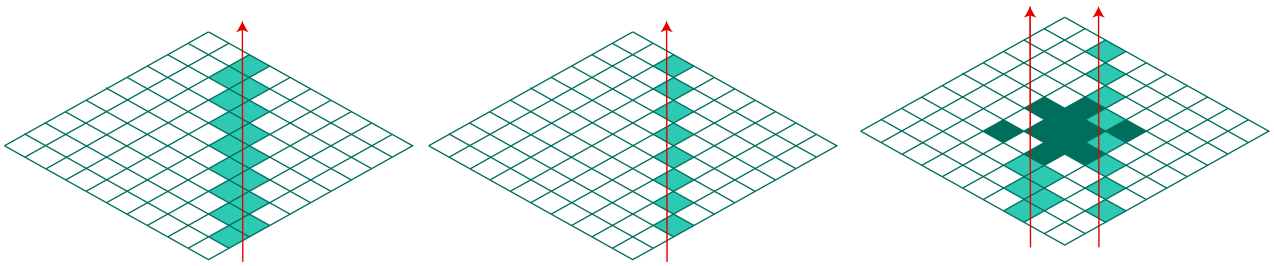


Figure 3.2: The ray-cone diverges as it progresses away from the camera



(a) Travelling one axis at a time (b) Travelling multiple axes at a time (c) Imprecise traversal (right) compared to more precise traversal (left)

Figure 3.3: Travelling multiple axes at a time can save up to 66% visits, but imprecise traversal may miss fragile geometry.

an infinitely thin line. By using a cone with the correct 'divergence', depending on the resultion and field-of-view, the resulting image is free of aliasing and moiré. While this technique is costy in raytracing, it is rather simple in discrete raycasting, given a the volume is in a *level-of-detail* (LOD) structure. In a level-of-detail structure (MIP maps and SVOs are examples of such structures), a filtered sample is availabe for every level of detail of the original image. So if the ray hits the volume near the camera, a completely refined voxel is taken, while far from the camera a less or unrefined voxel is used. Since every sample is a filtered average of lower levels, we instantly obtain the integral of the cone. Apart from being beneficial for the image quality, we gain in performance as the depth traversal may be terminated earlier.

An SVMM encoded voxelmap is only an LOD structure if $w = 2$, which will almost never be the case. Due to this fact, the principle of filtering using LOD can only be partly achieved. It can still improve image quality and performance, but we also believe that anti-aliasing should be implemented as a post-processing step. This post-processing can be done using, for example, a bilateral filter [ED04]. Using such a filter is a relatively small performance penalty, while it also can improve quality of object-background and object-object edges as well as the smoothing of ambient occlusion and normals - all of which the cones technique cannot.

3.2.5 Precision

The baseline algorithm uses incremental arithmetic on single-precision floating point numbers. On really large volumes this approach may eventually lead to artifacts. The work of Stolte et al. [SC95] proposed a method to alleviate this problem, although it was not designed for modern graphics hardware. Using anything other than floating points will severely degrade the performance on modern GPUs (some support double precision, but at half the speed). In our experiments, these artefacts were generally acceptable and almost removed by post-render anti-aliasing. Nevertheless, it is a cause for concern. One could also lower the precision further by changing lines 13-14 to use $\lfloor t_{max} \rfloor$. The algorithm may now pick multiple axes to traverse, making the traversal some 30% faster. See Figure 3.3. This optimization does however introduce noticeable artefacts.

3.3 Block memory management

The linear structure that the SVMM encoder uses to write individual blocks to the disk, cannot be used to store the data on the GPU. We can, however, exploit the 3D texture features of the hardware. By allocating a large enough 3D texture to form a *block pool*, it is possible to store all blocks from one level in their original order and still benefit from the spatial caching. In order to do this, one block pool per mipmap level is created. Because different levels can have a different block width, it would be impractical to store all those

differently sized blocks in a 3D structure. Secondly, a hash function is needed to translate the offset of a block to a set of texture coordinates.

Before a hash function can be created, the optimal size (U, V, W) of the pool must be established, given an amount of blocks N.

$$
\begin{aligned}
c &= \lfloor \sqrt[3]{N} \rfloor \\
U = V &= 2^{\lfloor \log_2 c \rfloor} && \text{Round } c \text{ to the nearest power of two} \\
W &= \lceil \frac{N}{U * V} \rceil && \text{The remaining space is allocated in the W-direction}
\end{aligned}
$$

By making U and V a power of two, the hash function can make use of arithmetic shifts instead of divisions. The waste of space that is inherent to the rounding is, when empirically tested, quite small for even billions of blocks.

Now a hash function $\mathbf{H}(o)$ can be defined that translates a block offset o within a certain level, to texture coordinates (u, v, w).

$$
\mathbf{H}(o) = \begin{cases}
u = o - (v + w) = o - \left\lfloor \frac{o - \lfloor \frac{o}{UV} \rfloor}{V} \right\rfloor - \left\lfloor \frac{o}{UV} \right\rfloor \\
v = \left\lfloor \frac{o - w}{V} \right\rfloor = \left\lfloor \frac{o - \lfloor \frac{o}{UV} \rfloor}{V} \right\rfloor \\
w = \left\lfloor \frac{o}{UV} \right\rfloor
\end{cases}
$$

The practical implementation of this function is very fast by means of incremental arithmetic and shift operations.

3.4 The rendering pipeline

The *geometry phase*, such as discrete raycasting in this case, is often part of a larger rendering pipeline. The raycasting as described in this chapter only deals with the color of a fragment based on its geometry (volume). Other steps may, for example, deal with reflections, lighting, refraction and anti-aliasing. Nowadays, it is increasingly common to implement these steps in *screen space*, which means that they are performed after the geometry phase on a per-fragment basis (sometimes also called *deferred rendering*). The advantage is that the complexity is given by the screen resolution and not by the resolution of the geometry, while the main disadvantage is that it is often an approximation of lesser quality. Often some additional data is required to enable screen-space calculations, such as the fragment's depth, normals or world position calculated during the geometry phase.

The implementation that is used for the evaluation of our work, uses a set of simple screen space rendering steps. Lighting can either be provided by a simple Phong-Blinn shader or screen space *ambient occlusion* (SSAO, see [BS08]). In addition to the LOD-based anti-aliasing provided by the MIP mapping, a very simple depth-based anti-aliasing is performed. To make any of these steps possible, however, fragment normals need to be approximated first.

3.4.1 Screen-space normal approximation

While outside the scope of this thesis, we feel the necessity to briefly discuss the problems we encountered related to normals in a (discrete) volumetric data set. The term *normal* refers to a vector that is perpendicular

to a certain surface, while also being of unit length and pointing 'out of the surface'. Normals are one of the basic building blocks in computer graphics and are used for almost any imaginable computation of 3D surfaces. In reality, a voxelized surface can only have normals pointing in the directions of the six faces of a cube. This results in a low image quality and unnatural lighting. Therefore, if we assume that the surface represented by the voxels is smooth, we can try to approximate this surface and calculate its normals.

An approach that works on continuous geometry, is to calculate each fragment's position from the depth buffer and then calculate its normal using the positions of its neighbouring fragments. Voxelized geometry is, however, inherently discrete and the depthbuffer is therefore discontinuous. See also Figure 3.4a. The depth buffer can be smoothed to a certain degree, by using the linear filtering mode of the hardware texture units. For every texel that is requested, the surrounding texels are also obtained and interpolated.

By using the interpolated depth, we can calculate the world-space positions of the surrounding fragments. We then use a circular sample pattern and construct triangles from the original fragment to the points on the circle. We repeat this while increasing the radii of the circles until (approximately) the size of a voxel is covered. We also check the depth of the samples agains the original fragment. If the difference is beyond some treshold, the sample is rejected in order to respect the occlusion boundaries. Figure 3.4b further demonstrates this principle.

For a complete survey of some more sophisticated methods, we refer the reader to [YCK92].

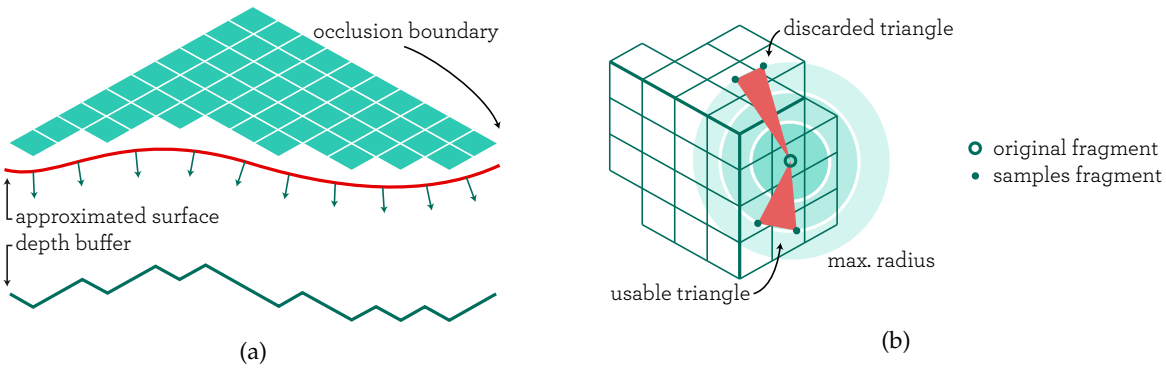


Figure 3.4: Approximation of surface normals from a discontinuous depth buffer

Chapter 4

Evaluation

In order to determine the effectiveness of the SVMM algorithm as well as to establish a comparison with other methods, it was subjected to an empirical evaluation. Our experimental implementation consists of the work flow mentioned in Section 1.1, including the networking part. The server was deployed on a large multi-cpu machine with 20 Intel Xeon E5-2650v3s, multiple video cards and 128 Gb of RAM. The renderer itself was tested on a Nvidia Geforce GTX Titan X with 12 Gb of VRAM. The client software ran on a recent MacBook Pro, either connected through the internet or through a high-speed university network.

Like the rest of this thesis, we decided to focus primarily on the efficiency of the SVMM encoding/decoding. Because of this and the fact that the network code is not mature enough at the time of writing, our evaluation only includes results from the SVMM tests. In these experiments, the goal was to assess its practical use given the following questions:

- Can a sufficient compression ratio be achieved to store a large voxelmap (2048^3) on a single GPU?
- Is the rendering performance comparable to a non-compressed voxelmap?
- How does compression ratio and performance compare to a basic SVO?
- What is the influence of the parameters of the SVMM encoder on performance and size?

4.1 Test data

Three data sets were used to create the test groups: a procedurally generated *Menger sponge* (`menger`), a microscopic scan of a zerbrafish (`fish`) and a microscopic scan of a fish eye (`eye`). Both `fish` and `eye` are in `DENSITY8` format. The `DENSITY8` voxel format is an 8 bit grayscale format where the intensity γ also determines the alpha channel in rgba $= (\gamma, \gamma, \gamma, \gamma)$. The Menger sponge was generated at multiple levels, colored with a RGB gradient and stored uncompressed to disk, in order to be able to compare it with the other data sets. All three sets were converted to a common uncompressed datastructure first. Additional details about the sets can be found in Table 5.2.

4.2 Test groups and measurements

To serve as a reference, all thee data sets were tested in two reference conditions: completely uncompressed (original data) and loss-less compressed using an 'SVO'. The latter condition was created by using the SVMM decoder with the parameters $w = 2, r = 2, \delta = 0$. While this is perhaps not a true SVO (more a SVO-MIP

map hybrid), it does create 2^2-sized octree blocks, a comparable tree-depth and no additional compression. The data was rendered automatically using simulated user input: rotate the camera once, zoom all the way in and all the way out. 94 frames of 1920×1080 were rendered for each separate condition.

The SVMM test groups consists of SVMM files compressed to disk with varying parameters to the encoder. The following parameters were used:

- Equal blockwidth for all levels of $w = 4$ and $w = 8$
- Root width of $r = 8$, $r = 32$ and $r = 128$ (if applicable)
- One of 'high' quality and one of 'low' quality (dependent on test data, determined individually)
- Different blockwidths for each level: level o being $w = 2$, level $n - 1$ being $w = 2^n$ (denoted as $w = 0$ in the results)

In addition to the test parameters listed above, the effect of block compression on the base level (as discussed in Section 2.1.1) was evaluated. A very basic scheme was used were groups of 2^3 voxels are compressed to a bitmap (more specifically, to a 8 bit integer) and their color is determined by one collective RGB8 or DENSITY8 value (see Figure 1.2c). Arguably, a more sophisticated encoding such as S3TC could be used, but for our prime interests (size and performance) this would not make a difference.

For the evaluation of performance, three magnitudes of measurement were selected: compression ratio, GPU memory usage and time to render one frame (average, min, max). For the latter we strictly measured the time it took to run the raycast kernel. The transfer of the volume to the GPU, the post-processing filters and the read-out of the framebuffer are not included in this figure. To put this timing in perspective, a frames-per-second based on the *total* rendering time is also included.

Rounded averages were used to determined the render time for a single frame. To this end, CUDA's internal 'event' measurement functions were used as they provide up to nanosecond precision and are specifically tailored to the use with CUDA kernels. All timings were performed by rotating the camera once around the data set and then zooming the camera in all the way and back again.

4.3 Results

Table 5.1 shows the results from every test group. For the sake of brevity, in the SVMM group only the 'best' results and their parameters are shown. This 'best' result was determined by only looking at the high-quality ($q \geq 80$) samples, while trying to balance size and performance. In Figure 5.2, however, all parameter combinations were plotted against the two main measurements of performance (time and ratio). The points in the diagrams are labelled with the parameter triplets in the form (w, r, q). Where $w = 0$ is written, we mean that a stepped blockwidth was using (starting at $w = 2$ steps in powers of two for each next level). Figures are provided for each data set.

Figure 5.3 shows the number of mipmap level against the raycast time. In order to investigate a possible trend, a curve was fitted using Non-Linear Least Squares fitting. The Figure shows only the `menger` sets as the `fish` and `eye` sets did not show such behaviour.

Chapter 5

Discussion

From the results can be seen that SVMM can offer a substantial rate of compression, while limiting the amount of used mipmap levels. When compared to uncompressed voxelmaps, both performance improve and size is reduced by at least 50%. The increase of speed could be explained by the skipping of empty space and the use of camera distance to determine level-of-detail. The performance improved slightly with the larger sets, and in addition, the very large set cannot even be rendered at all when not in some way compressed. For really large volumes of 2048^3 and over, there is currently no other way to utilize a single GPU with any means of compression. The proposed algorithm provides a very adequate performacne here.

Compared to our 'SVO' it can be concluded that SVMM offers better raycast performance while SVO offers slightly superior loss-less compression ratios. The prime reason for this is that the blockwidth of 2 provides smaller blocks which lead to more compression, but also to deeper trees. Finding the 'neighboring voxel' is the most time consuming part of the raycaster. When using an (simplified) octree, this problem has a complexity of $\log n$ as a function of the depth of the tree. Thus reducing this depth can, in whatever way, greatly improve performance.

There are other ways to improve performance without reducing the depth of octree. For example, Gobbetti et al. [GMG08] use a spatial index structure to precalculate every voxel's neighbors. This method would use slightly more GPU memory but also significantly increase the complexity of the encoder, which may or may not be desireable for a certain application. We believe, however, that the slightly worse efficiency of the SVMM loss-less compression performance can be offset by the fact that our algorithm can use lossy compression to enhance the compression ratio significantly.

When block compression is enabled, the ratios improve significantly: up to 100:3 depending on the data. Fortunately, the currently employed compression scheme does not seem to add any performance penalty. It would be interesting to see if this also hold for more sophisticated schemes.

It should probably be noted that the render times are largely similar in all data sets. This result cannot be entirely natural, as the raycast performance is mainly dependend on the size of the data. Two explanations can be offered: both `fish` and `eye` are very shallow in one direction, therefore raytraversal terminates quickly. Secondly, something similar could be said about the `menger` sets: roughly $\frac{2}{3}$ of the surface of the Menger-sponge is simply a cube. On these portions, raycasting terminates immediately, dramatically influencing the mean render time. Further study should therefore be done using different data sets.

5.1 Tested parameters

Judging from Table 5.1 and Figure 5.2, it is hard to predict the effectiveness of a set of parameters. For example, by changing the block width by one step, can have a profound effect on the final size. Moreover, parameters that work on one data set, result in less-than-optimal compression in others. A trend can be discovered, however, by comparing the optimal results from Table 5.1. The following statements about possible correlations can be made:

- Stepped blockwidth ($w = 2, 4, 16, 32$ etc.) appears to have the best result for non-block compressed samples in every case
- The larger models seem to benefit from an increased r
- For the block-compressed samples, no correlation between w and the final ratio seems to exist

It would seem that further research is in order to establish a heuristic for these parameters. Moreover, a larger range of values should be tested.

5.2 Image quality and compression

The quality of an image is somewhat subjective and also hard to quantify. When using lossy compression, one may wonder how much of the original data is lost and if this impairs the usability of the visualisation. During the experiments that were conducted, the image quality was carefully evaluated and almost no visible degradation was detected using the parameters in Table 5.1. However, an interesting observation can be made from Figure 5.4. The original `eye` data set can be reduced from 193 Mb to 91 Mb with $q = 90$. The loss in detail is minimal. Reducing the size further to 41 Mb by $q = 65$, however, significantly deteriorates the image. On the other hand, by applying block compression and using $q = 90$, the image is not degraded any further while the size is reduced to only 14 Mb. It seems that on 'fragile' data with subtle transitions, such as `eye`, benefits from block compression are more appearant than those from the MIP map-based compression. We have seem similar results, albeit less pronounced, on the solid geometry of the `menger` sets.

5.3 Correlation of depth and performance

In the introduction we theorized about a possible correlation between the number of mipmap levels (or the depth of the octree) and the raycast performance. As shown in Figure 5.3, this effect appears to exist for the `menger` data sets. There is, however, too little statistical evidence that such correlation really exists even though the results do point in that direction. The fact that the other two data sets did not show the same behaviour, may be due to the limited resolution of these data sets in one direction. Because of this limited resolution of one axis, the depth in that direction is virtually constant and no correlation could be shown as such.

5.4 Future work

The workflow project as presented is still in its infancy. For example, the tethered rendering part could be a thesis entirely on its own. As far as the rendering and compression part is concerned, there is still much unexplored territory. The following paragraphs provide a brief overview of possible subjects left to pursue.

5.4.1 Out-of-core rendering

Out-of-core rendering or streaming is a technique to keep a very small working data set ready on the GPU, while the rest of the volume is kept *out-of-core* on the disk. By identifying the parts that are visible from the current viewpoint, a working set can be transferred from the host to the GPU. In this way, a potentially infinitely large data set could be rendered, provided it is not entirely visible at once. For non-transparent models only a few percent of the voxels is visible at a given time. The works by Gobbetti et al. [GMG08] and Crassin [Cra11] have successfully implemented such a system and are probably among the best academic solutions available.

The SVMM algorithm could be modified to support streaming by introducing *spatial regions*. Spatial regions are complete, self contained SVMM datastructures that encode a fraction of the original volume. The root block is augmented with an index to one of these regions. The renderer can request any of these regions from the CPU, up to a certain depth. Therefore, from a large distance many regions will be loaded with little depth, while for a close up few regions will be loaded with full depth. A least-recently-used cache policy could be implemented to asynchronously replace blocks between two frames. Such system a would add some overhead, but also provides a means of rendering extremely big data.

5.4.2 Wavelet compression

Wavelet compression, as discussed in Section 1.5.4, is a means of lossy compression of blocks of data. It uses a transformation to reduce entropy after which compression can be more effective. Wavelet compression is already used successfully in volume compression, for example by Kim et al. [KS99]. While it is not feasible to apply wavelet transformation on an entire volume (it has to be in memory or duplicated on disk), wavelet compression could be used to replace or augment the block compression currently used for our SVMM encoder.

5.4.3 Segmented raycasting

While raycasting is *massively parallel* by nature, the ray traversal algorithm scales poorly on GPU. For example, occlusion culling using early ray termination aborts the traversal at the point where occluded voxels are encountered. While this gives a massive performance improvement on the CPU, threads that abort traversal are simply stalled until the entire warp is finished on the GPU. By traversing a ray in multiple segments, either sequentially or in parallel, the execution time of the whole warp will become shorter, leading to less stalls. We implemented this theory on our raycaster and found only adverse effects on performance. This is probably due to the redundant nature of most of the ray segments. We believe, however, that in future research this method could be perfected up to a point where it actually improves render times. A somewhat similar idea was demonstrated by Stolte et al. [SC95], albeit in the pre-GPGPU era.

5.4.4 Rendering by proxy

The datastructures used in any volume compression scheme, including our own, are difficult to port to the GPU due to register usage and branching. Moreover, the ray traversal relies heavily on random, non coalesced accesses to global memory, which may well be the Achilles' heel of modern GPUs. Although 3D texture memory alleviates this problem to a large extent, it would also be possible to relocate some of this work to the CPU. Our hypothesis is that by using a bitmapped *proxy* version of the volume and storing the coordinates to the framebuffer instead of the color values, the CPU could be used to fill in the color values in *screen space*.

Traversing a bitmapped volume costs significantly less memory and memory accesses. In addition to this, a very simple algorithm can be used to traverse it. The CPU-side 'coloring' could be performed asynchronously and by doing this in screen-space, its complexity is greatly reduced. In the meantime, the GPU could already calculate normals, perform SSAO and possibly other filters.

5.5 Conclusion

We have proposed a datastructure for the lossy compression of volumetric data. These Sparse Voxel MIP Maps (SVMM) are based on a combination of 3D MIP mapping, Sparse Voxel Octrees (SVO) and block compression. This combination enables both lossy compression an support for a variety of voxel formats. By using block sizes that can be individually set per mipmap level, the depth of the 'tree' is reduced while storage sacrifices are small. Lossy compression is achieved by removing blocks with little information and by applying additional block compression to the base-level.

Apart from the compression algorithm, a raycast implementation in CUDA was demonstrated to decode and render the datastructure efficiently. In order to utilize the beneficial features of hardware 3D texture units, a block management method was proposed to store the (linear) mipmap levels in separate 3D textures. Furthermore, multiple techniques were introduced that can be used to improve the performance of the baseline raycast algorithm.

We have shown that the proposed algorithm can be used to provide favorable compression ratios (up to 100:3) while remaining good rendering performance. The parameters that lead to these compressions, are still a little unpredictable and benefit from tuning on an individual basis. In comparison to traditional SVOs, SVMM is able to provide a better balance between size, performance and quality. Moreover, for all data sets, the SVMM is faster than its uncompressed counterpart.

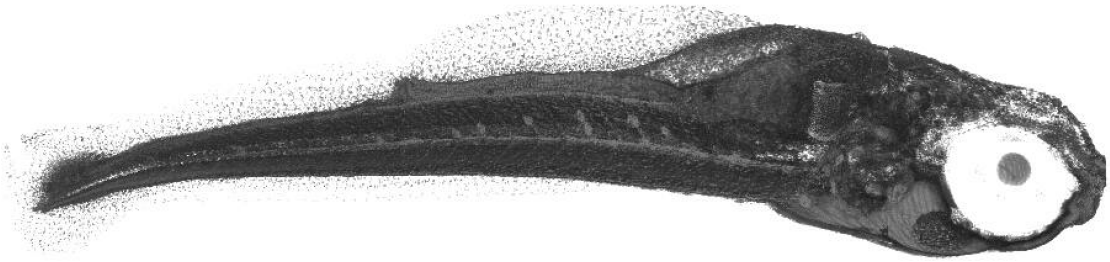
The use of block compression seems to be extremely useful on volumetric data. The best compression ratios were achieved with block compression and the image degradation was shown to be minimal. Additionally, block compression gives a very little, if any, reduction of performance of the raycaster.

As stated in the introduction, the rendering and exchange of volumetric data could benefit from a lossy compressed format. SVMM may be a good attempt at this and perhaps, with the adjustments suggested in Section 5.4, it could very well be a practical option.

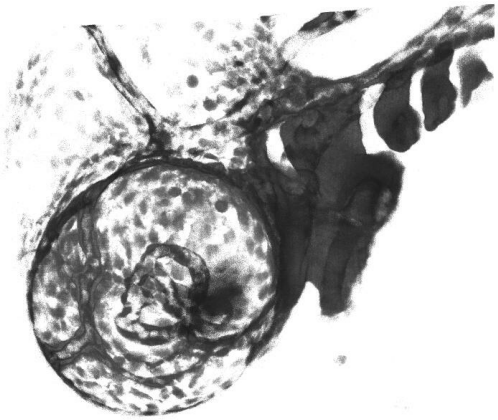
Appendix A - Results

Test group		Parameters				Results		
Group	Data set	w	r	quality	Ratio	Size (GPU)	Time (raycast)	Time (total)
Uncompressed	Fish				1:1	346 MB	2.098 ms	464 fps
	Eye				1:1	358 MB	2.422 ms	566 fps
	Menger-5				1:1	228 MB	2.180 ms	470 fps
	Menger-6				1:1	1687 MB	3.583 ms	298 fps
	Menger-7				1:1	too large	-	-
SVO	Fish				100:22	215 MB	1.726 ms	578 fps
	Eye				100:28	229 MB	1.763 ms	566 fps
	Menger-5				100:45	205 MB	3.636 ms	281 fps
	Menger-6				100:33	691 MB	2.695 ms	392 fps
	Menger-7				100:25	11763 MB	1.646 ms	153 fps
SVMM	Fish	0	32	90	100:17	199 MB	1.551 ms	644 fps
	Eye	0	8	90	100:27	299 MB	1.559 ms	639 fps
	Menger-5	0	32	80	100:47	198 MB	1.539 ms	647 fps
	Menger-6	0	32	80	100:34	700 MB	1.492 ms	666 fps
	Menger-7	0	128	80	100:25	11915 MB	1.513 ms	155 fps
SVMM, block encoded	Fish	4	8	90	100:3	177 MB	1.549 ms	644 fps
	Eye	8	32	90	100:6	181 MB	1.545 ms	647 fps
	Menger-5	4	32	80	100:8	175 MB	1.496 ms	665 fps
	Menger-6	8	32	80	100:6	284 MB	1.492 ms	666 fps
	Menger-7	4	128	80	100:4	2902 MB	1.516 ms	155 fps

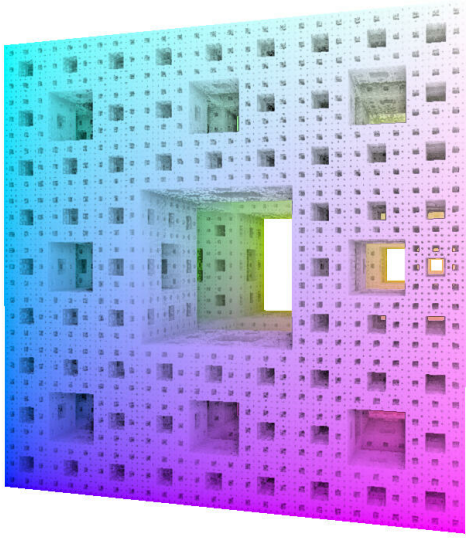
Table 5.1: Results



(a) The `fish` data set



(b) The `eye` data set

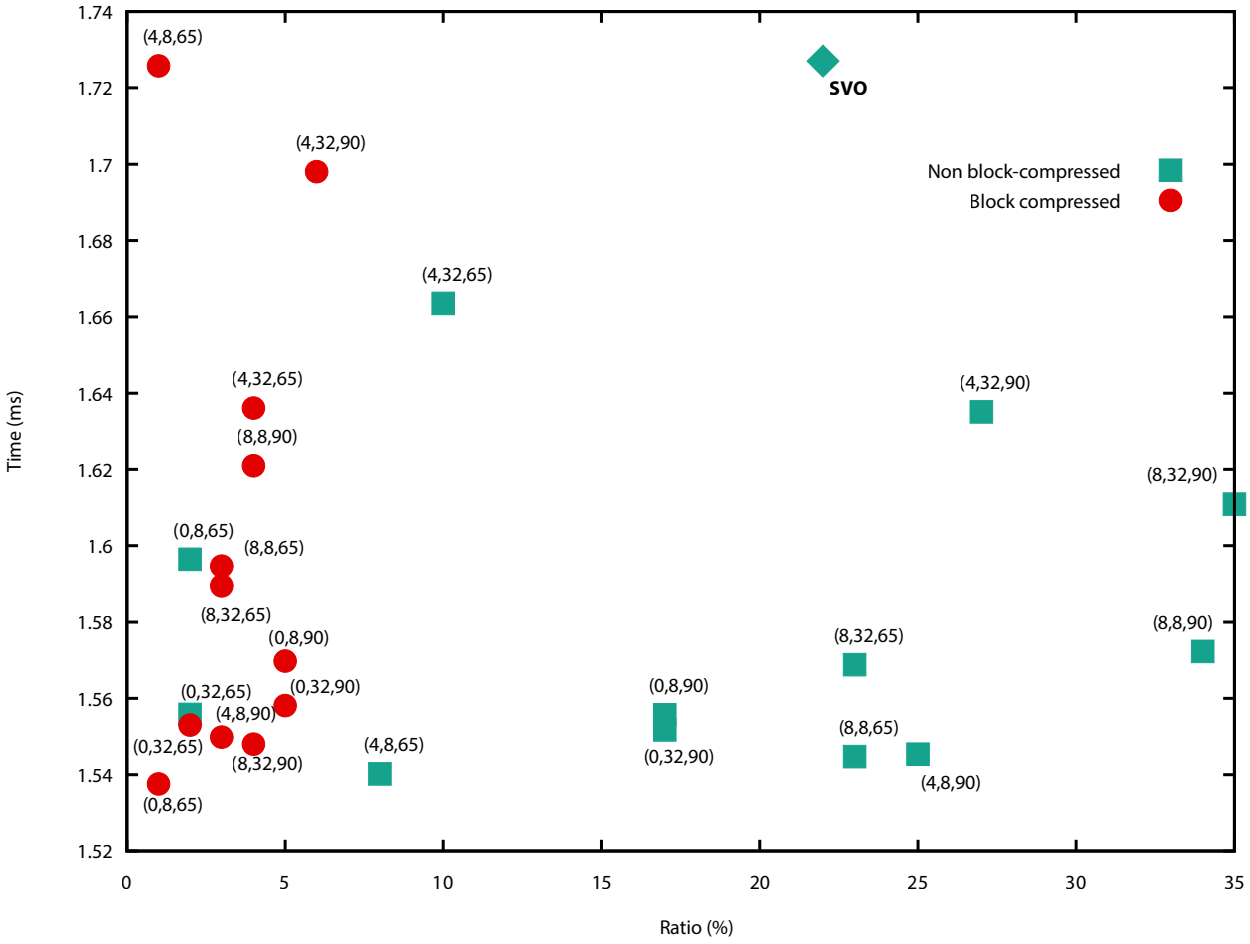


(c) The `menger-6` data set

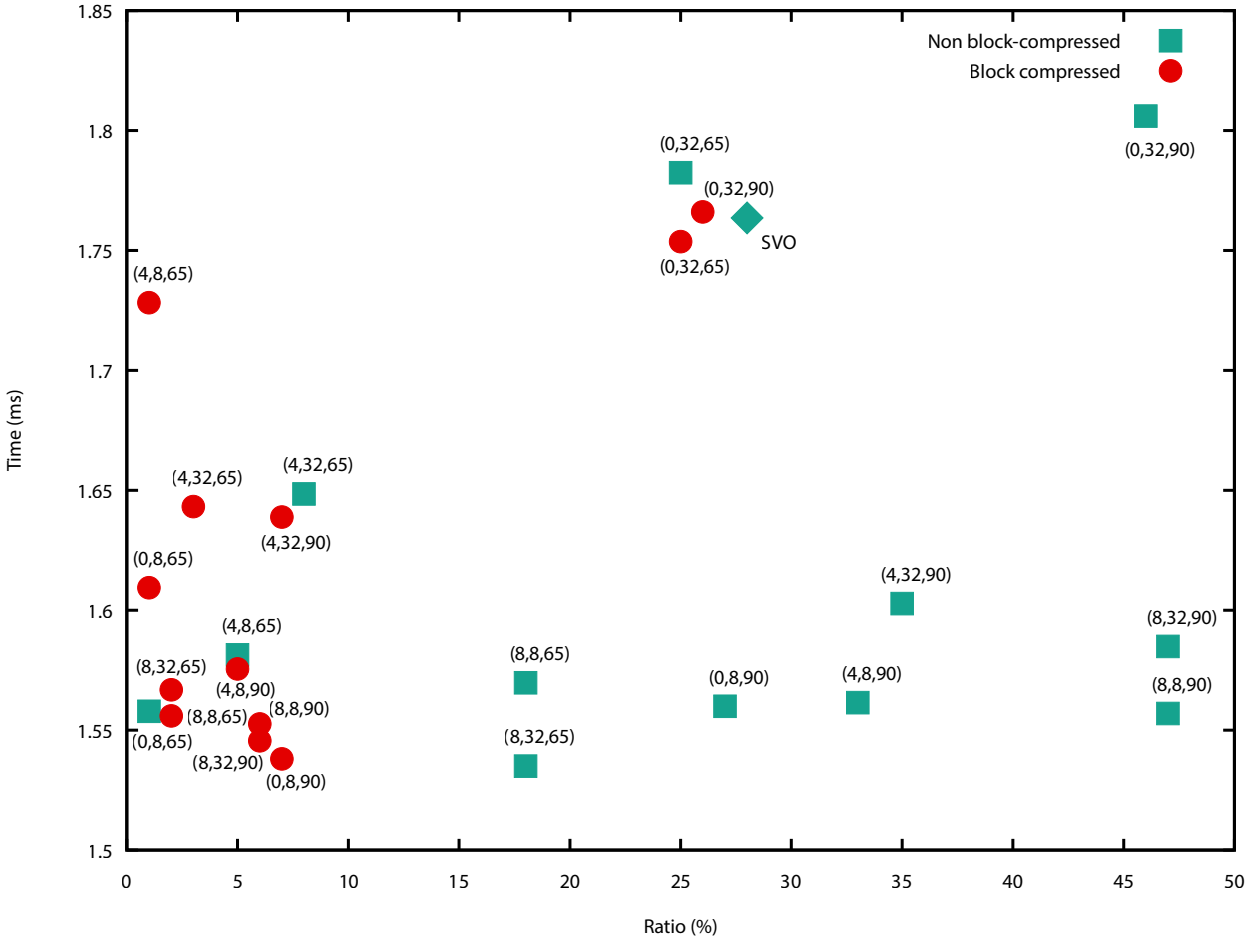
Figure 5.1: Data sets used for evaluation

Set	Volume size	Voxel format	Uncompressed size
Fish	$3072 \times 1024 \times 72$	`DENSITY8`	163 Mb
Eye	$2048 \times 2048 \times 48$	`DENSITY8`	193 Mb
Menger-5	$243 \times 243 \times 243$	`RGBA32`	55 Mb
Menger-6	$729 \times 729 \times 729$	`RGBA32`	1,5 Gb
Menger-7	$2187 \times 2187 \times 2187$	`RGBA32`	39 Gb

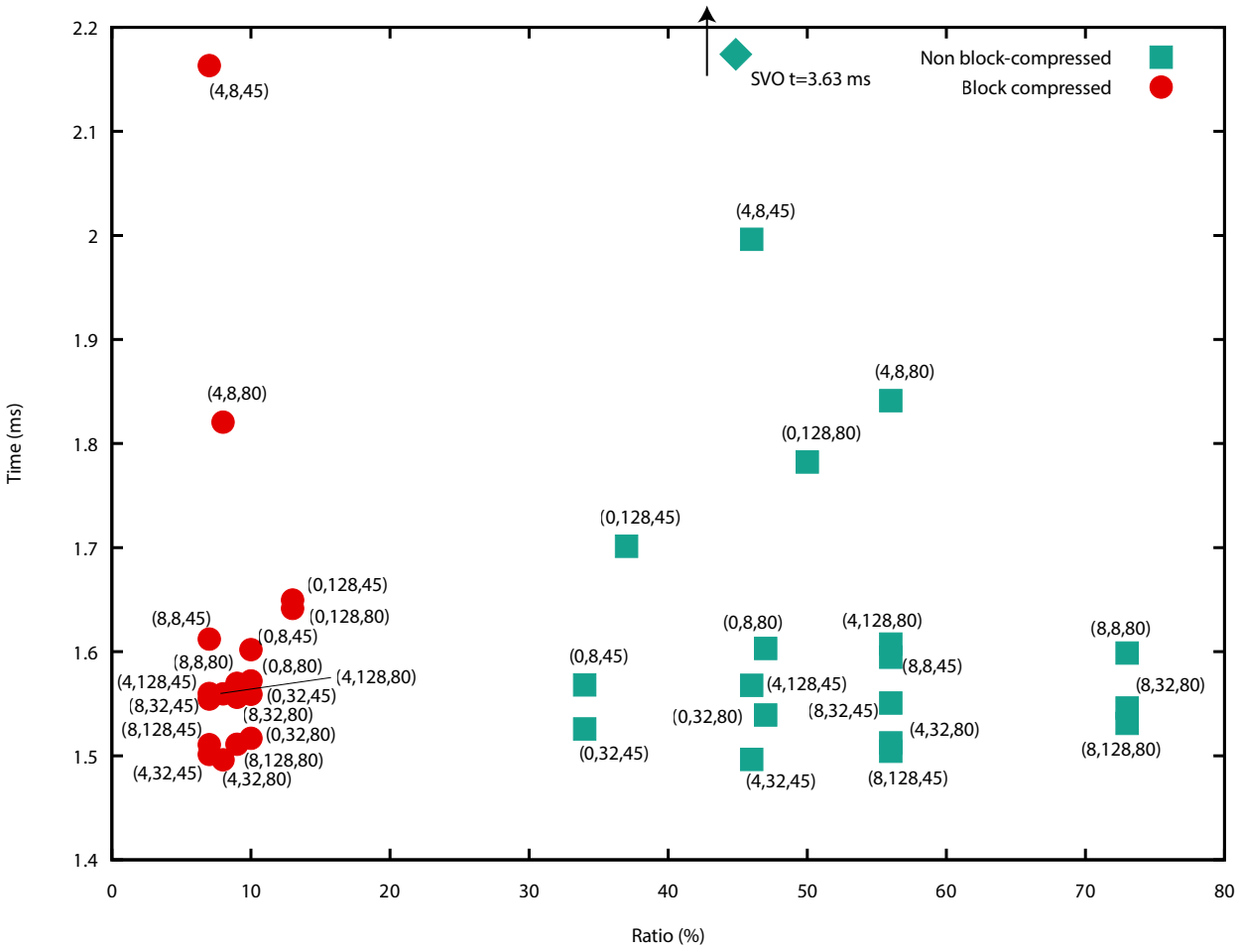
Table 5.2: Data sets and properties



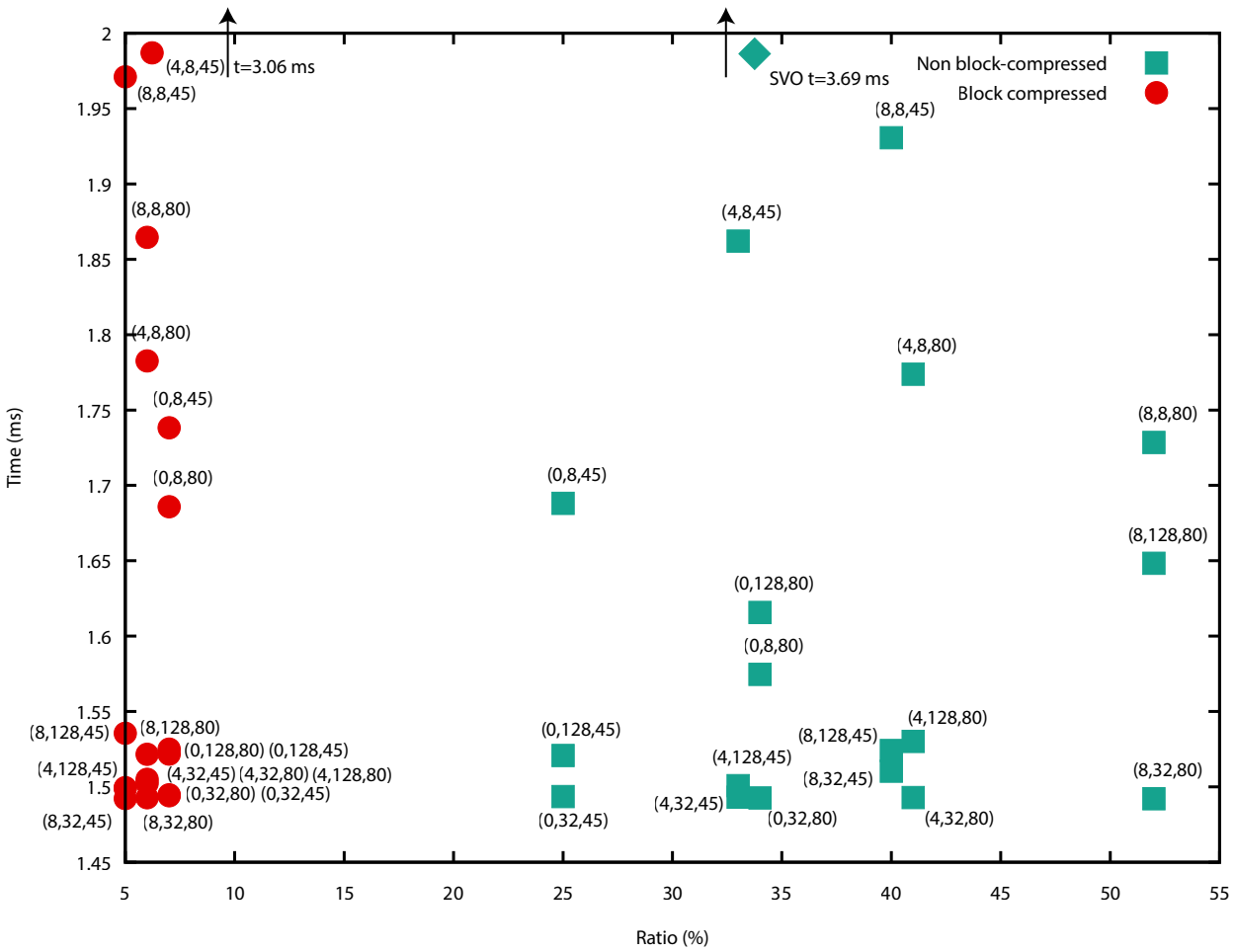
(a) Results for `fish`



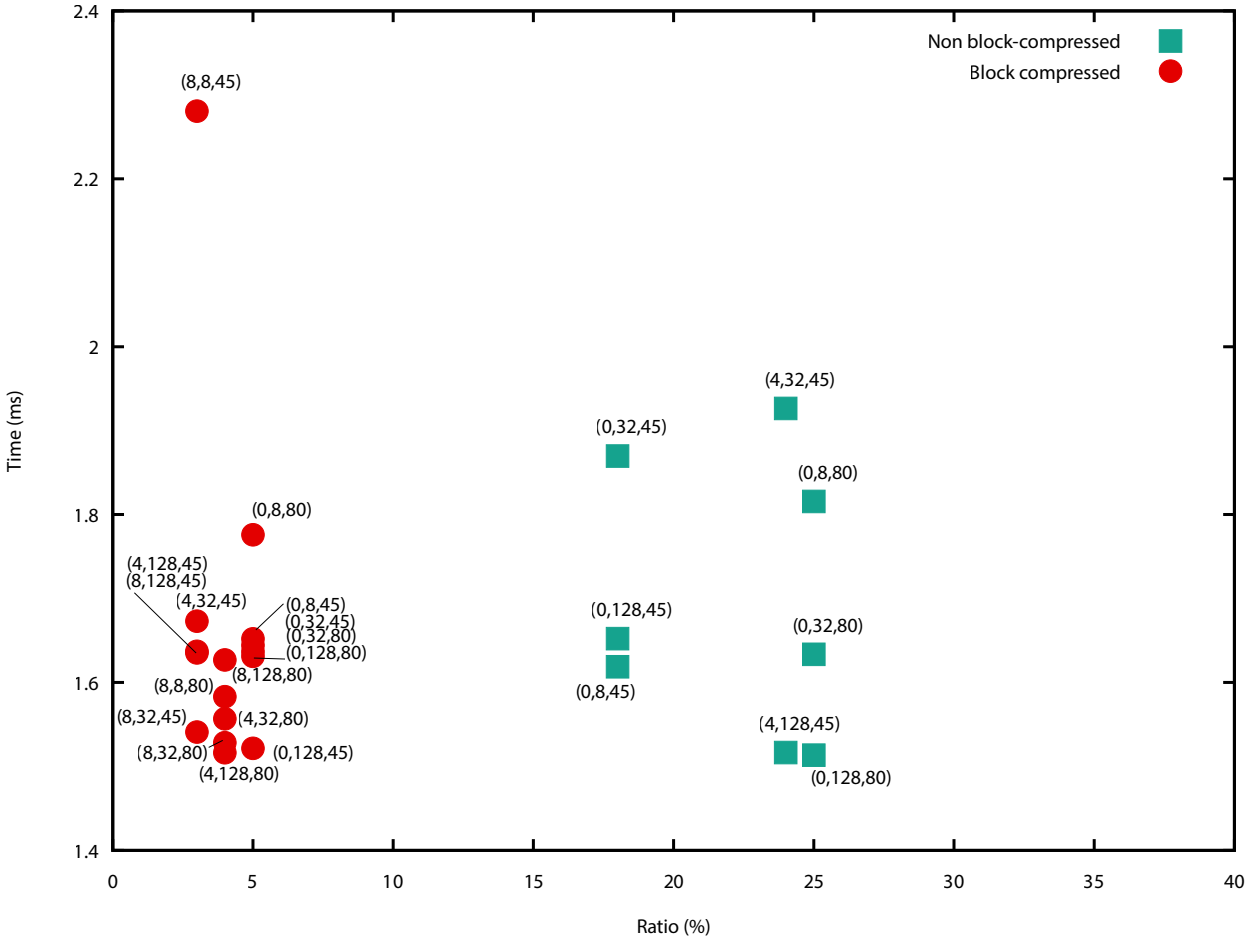
(b) Results for `eye`



(c) Results for `menger-5`



(d) Results for `menger-6`



(e) Results for `menger-7`

Figure 5.2: Results per data set

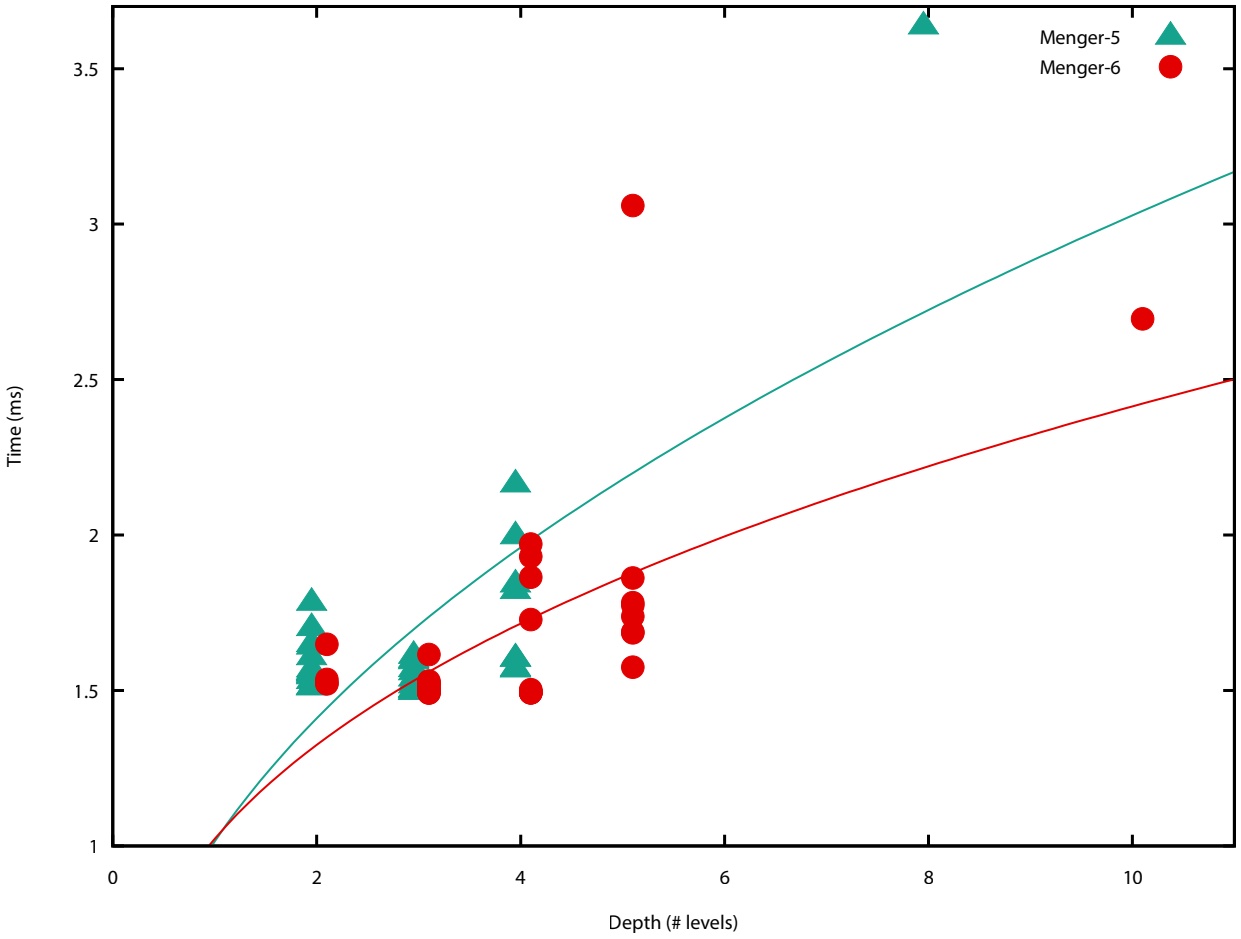
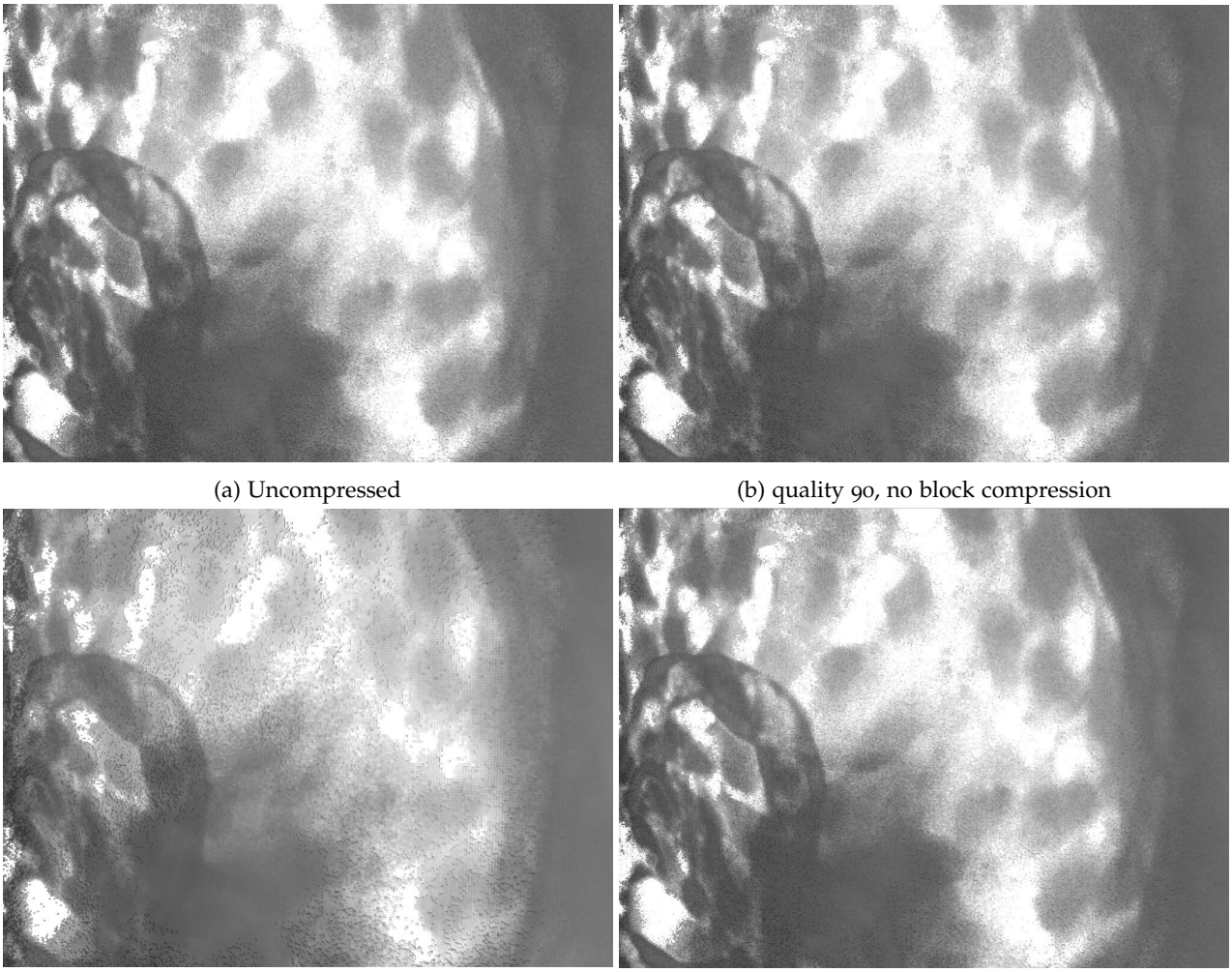


Figure 5.3: Number of mipmap levels against raycast time.



(a) Uncompressed

(b) quality 90, no block compression

(c) Quality 65, no block compression

(d) quality 90 with block compression

Figure 5.4: Different degrees of image degradation in a close-up of `eye`

Bibliography

[Ama84] John Amanatides. Ray tracing with cones, 1984.

[AW87] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. *Eurographics*, 3, 1987.

[Bar11] Tavian Barnes. Fast, branchless ray/bounding box intersections. website, 2011. visited August 2016.

[Bre65] Jack E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[BS08] Louis Bavoil and Miguel Sainz. Screen space ambient occlusion. Technical report, NVidia Corporation, september 2008.

[Cra11] Cyril Crassin. *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, L'UNIVERSITE DE GRENOBLE, 2011.

[Cro77] Franklin C. Crow. The aliasing problem in computer-generated shaded images. In *Communications of the ACM*, page 799805, November 1977.

[ea89] A. Glassner et al. *An Introduction to Ray Tracing*, chapter Ray - Box intersections. Academic Press, 1989.

[ED04] Elmar Eisemann and Frédo Durand. Flash photography enhancement via intrinsic relighting. *ACM transactions on graphics (TOG)*, 23(3):673–678, 2004.

[Eo7] Alphan Es and Veysi Is ler. Accelerated regular grid traversals using extended anisotropic chessboard distance fields on a parallel stream processor. *Journal of Parallel and Distributed Computing*, 2007(67):1201–1217, 2007.

[FI85] A. Fujimoto and K. Iwata. Accelerated ray tracing. *Proc. CG Tokyo*, pages 41–65, 1985.

[FO11] S. Forstmann and J. Ohya. Efficient, high-quality, gpu-based visualization of voxelized surface data with fine and complicated structures. *IEICE TRANSACTIONS on Information and Systems*, 2011.

[FVS11] J. Fang, A. L. Varbanescu, and H. Sips. A comprehensive performance comparison of cuda and opencl. In *2011 International Conference on Parallel Processing*, pages 216–225, Sept 2011.

[GMG08] Enrico Gobbetti, Fabio Marton, and Antonio José Iglesias Guitián. A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7):797–806, 2008.

[KS99] T. Kim and Y. Shin. An efficient wavelet-based compression method for volume rendering. *Proceedings. Seventh Pacific Conference on Computer Graphics and Applications*, pages 147 – 156, 1999.

[Lev90] M. Levoy. Display of surfaces from volume data. *IEEE Comput. Graphics*, 9(3):245–261, 1990.

[LK10] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *Proceedings of I3D 2010*, 2010.

[RGG+14] M. Balsa Rodrguez, E. Gobbetti, J.A. Iglesias Guitin, M. Makhinya, F. Marton, R. Pajarola, and S.K. Suter. State-of-the-art in compressed gpu-based direct volume rendering. *Computer Graphics Forum*, (33):771–100, 2014. doi: 10.1111/cgf.12280.

[RLoo] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pages 343–352. ACM Press/Addison-Wesley Publishing Co., 2000.

[S3] S3 Corp., http://www.s3.com/savage3d/s3tc.pdf. *S3TC DirectX 6.0 Standard Texture compression*.

[SC95] Nilo Stolte and Rene Caubet. Discrete ray-tracing of huge voxel spaces. *EG Computer Graphics Forum*, 1995.

[SL10] Nicholas Schwarz and Jason Leigh. Distribution volume rendering for scalable high-resolution display arrays. In *GRAPP- International Conference on Computer Graphics Theory and Applications*, pages 211–218, 2010.

[SM02] Jon Sweeney and Klaus Mueller. Shear-warp deluxe: The shear-warp algorithm revisited. In *Joint Eurographics-IEEE TCVG Symposium on Visualization 2002*, pages 95–104, 2002.

[Sun91] Kelvin Sung. A dda octree traversal algorithm for ray tracing. In *Eurographics*, volume 91, pages 73–85, 1991.

[THGM11] Sinje Thiedemann, Niklas Henrich, Thorsten Grosch, and Stefan Muller. Voxel-based global illumination. *I3D Symposium on Interactive 3D Graphics and Games*, 2011.

[Wat94] Andrew B. Watson. Image compression using the discrete cosine transform. *Mathematica Journal*, 4(1):81–88, 1994.

[wav99] *Fractal and Wavelet Image Compression Techniques*. SPIE Press, 1999.

[WBMS05] Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. An efficient and robust raybox intersection algorithm. *Journal of Graphics Tools*, 2005.

[YCK92] Roni Yagel, Daniel Cohen, and Arie Kaufman. Normal estimation in 3 d discrete space. *The visual computer*, 8(5-6):278–291, 1992.